JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2005; **17**:27–64
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr.305

Research

Automating regression testing for evolving GUI software‡

Atif Memon[1,2,*,†], Adithya Nagarajan[2] and Qing Xie[2]

[1]*Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, U.S.A.*
[2]*Department of Computer Science, University of Maryland, College Park, MD 20742, U.S.A.*

SUMMARY

With the widespread deployment of broadband connections worldwide, software development and maintenance are increasingly being performed by multiple engineers, often working around-the-clock to maximize code churn rates. To ensure rapid quality assurance of such software, techniques such as 'nightly/daily building and smoke testing' have become widespread since they often reveal bugs early in the software development process. During these builds, a development version of the software is checked out from the source code repository tree, compiled, linked, and (re)tested with the goal of (re)validating its basic functionality. Although successful for conventional software, smoke tests are difficult to develop and automatically re-run 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, smoke-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 re-test 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 empirically compare the relative costs of employing different levels of detail in the GUI test oracle. We also show the events and statements covered by the smoke test cases. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: GUI regression testing; GUI testing; smoke testing; daily/nightly builds; event-flow graphs; graphical user interfaces; software quality

1. INTRODUCTION

Today's competitive software development market demands that several developers, who are perhaps geographically distributed, work simultaneously on large parts of the code during maintenance.

*Correspondence to: Atif Memon, Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, U.S.A.
†E-mail: atif@cs.umd.edu
‡A preliminary report of this work appeared in the *Proceedings International Conference on Software Maintenance* [1].

Received 24 December 2003
Revised 16 April 2004
Accepted 30 April 2004

Although successful for improving code churn rates, these practices also create new challenges for quality assurance, requiring the development of novel and practical solutions. One challenge is to provide rapid feedback to the developers about parts that may have inadvertently broken during maintenance. One approach to handle this challenge is to perform 'nightly/daily builds and smoke tests' [2–4]. Smoke tests have become widespread [5] as many software developers/maintainers find them useful [6]. Popular software that use daily/nightly builds include *WINE* [7], *Mozilla* [8], *AceDB* [9], and *openwebmail* [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* [5] and sometimes *acceptance tests* [12] are executed during smoke testing. Such tests are run to (re)validate the basic functionality of the system [11]. The smoke tests exercise the entire system; they do not 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 to be 'a good build'. Bugs are usually reported in the form of e-mails to the developers [5], who can quickly resolve the bugs. Frequent building and re-testing is also gaining popularity because new software development processes (such as extreme programming [13,14]) advocate a tight development/testing cycle [15]. A number of tools support daily builds; some of the popular tools include *CruiseControl* [16], *IncrediBuild* [17], *Daily Build* [18], and *Visual Build* [19].

A limitation of current nightly builds is inadequate testing and re-testing of software that has a graphical user interface (GUI)[§]. Frequent and efficient re-testing of conventional software has leveraged the strong research conducted for automated *regression testing* [21], which is a software maintenance activity, done to ensure that modifications have not adversely affected the software's quality [21]. Although there has been considerable success in developing techniques for regression testing of conventional software [22,23], regression testing of GUIs has been neglected [24]. Consequently, there are no automated tools and efficient techniques for GUI regression testing [25,26].

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 software systems. GUIs today constitute as much as 45–60% of the total software code [27]. Currently, three popular approaches are used to handle GUI software when performing nightly builds. The first, and most popular, is to perform no GUI smoke testing at all [11], which either leads to compromised software quality or expensive GUI testing later. The second approach 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 [28]), it also does not perform testing of the end-user software. The third approach is to use existing tools to do limited GUI testing [29,30]. Examples of some tools used for GUI testing include extensions of *JUnit* such as *JFCUnit*, *Abbot*, *Pounder*, and *Jemmy Module*[¶] and capture/replay tools [31] such as WinRunner[‖] that provide very little automation [32], especially for *creating* smoke tests. Developers/maintainers who employ these tools typically come up with a small number of smoke tests [3].

[§]Note that we focus on testing the functionality of the GUI, not *usability* [20] issues such as user-friendliness.
[¶]See http://junit.org/news/extension/gui/index.htm.
[‖]See http://mercuryinteractive.com.

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* [33]), test-case generation [34,35], test-oracle creation [36], and code instrumentation to test execution, coverage evaluation [37], 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 empirically compare the relative costs (in terms of time and space) of employing different levels of oracle information for re-testing. We also show the events and statements covered by the smoke test cases.

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 cannot only be used for nightly builds, but also for general GUI re-testing.
- We show our re-testing process as a natural extension of our already implemented GUI testing tools [24,25,32,34–37].
- We provide relationships between code and event coverage in the form of reports, to be used later during the testing phase.

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. In Section 4, we describe the GUI representation that enables us to perform efficient smoke testing. We then discuss the modules of DART in Section 5. The results of the experiments in Section 6 show that DART is efficient enough for frequent re-testing and produces coverage reports that can later be re-used for the testing phase. We discuss related research and practice in Section 7 and finally conclude in Section 8 with a discussion of ongoing and future work.

2. THE DART PROCESS

A very high-level overview of the DART maintenance process is shown in Figure 1. The top part of the figure shows the one-time `SETUP` phase, and the lower part shows the `ITERATIVE` nightly smoke testing cycle. During the `SETUP` phase, version i of the application is automatically analyzed, and test cases and oracles are automatically generated and stored for the iterative phase. As the application is maintained to produce version $i + 1$, smoke test cases are executed automatically. Reports, including bug and coverage reports are sent to the developers after smoke testing. The developers then fix the bugs, add new functionality, and the iterative smoke testing cycle continues.

We now present more details of the process as steps. The goal is to provide the reader with a step-by-step picture of the operation of DART during maintenance 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 I.

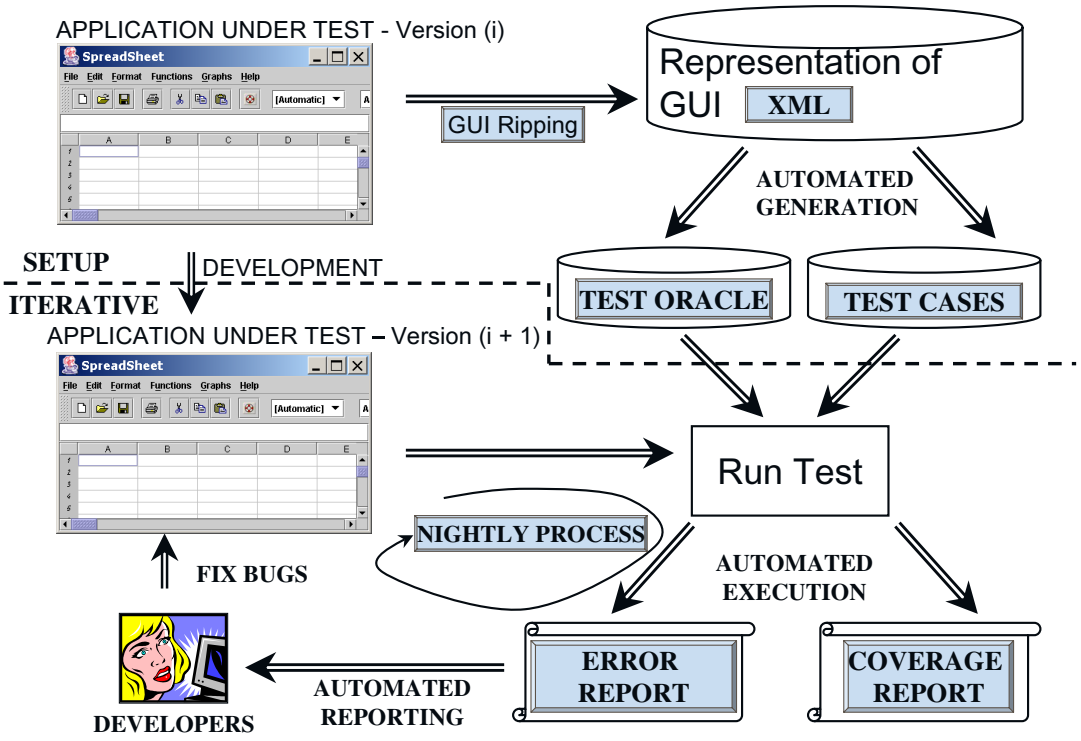

Figure 1. The DART process at a glance.

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* (Section 5.1). 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 required 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* [37] (Section 3). These structures are used in the next step and in step 7 to automatically generate test cases and evaluate test coverage.

Table I. Roles of the developer/tester and DART.

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

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 Table II. 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 Table III. Note that, in the matrix shown, the test designer has chosen to generate only a few length 3 test cases indicated in column 3, and none of the length 4 test cases, indicated by zeros 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.

Table II. Matrix M for MS WordPad.

	Test case length			
Component name	1	2	3	4
Main	59	791	14 354	255 720
FileOpen	10	80	640	5120
FileSave	10	80	640	5120
Print	12	108	972	8748
Properties	13	143	1573	17 303
PageSetup	11	88	704	5632
FormatFont	9	63	441	3087

Table III. Matrix M' for MS WordPad.

	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

9. The development team modifies the AUT during maintenance.
10. The operating system's task scheduler launches DART, which in turn launches the AUT. DART automatically instruments the AUT's source code and events. A code instrumenter (e.g., Instr [38]) is used to instrument the code whereas an event-level instrumenter (Section 5.5) is used to instrument the events. 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 (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 then modify M' accordingly.
17. The new test cases are generated.
18. The expected output for the test oracle is generated.

Steps 10–18 are repeated throughout the maintenance 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 maintenance cycle to improve overall test effectiveness and coverage.

3. DESIGN OF DART

Before we discuss the details of the design of DART, we 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 reports, and error reports) 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 2 shows the primary modules of DART and their interaction. The GUI representation is the 'glue' that holds all the modules together. The GUI ripper employs a new reverse engineering technique to automatically obtain parts of the representation. The test-case generator uses the representation to create GUI test cases. The test-oracle generator creates the expected state of the GUI to be used during testing. The code/event instrumenter instruments the code to collect coverage information during test execution. The test executor runs all the test cases on the GUI automatically and uses the coverage evaluator to determine how much testing was done. All of these modules interact with each other via the representation that this described next.

4. 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.

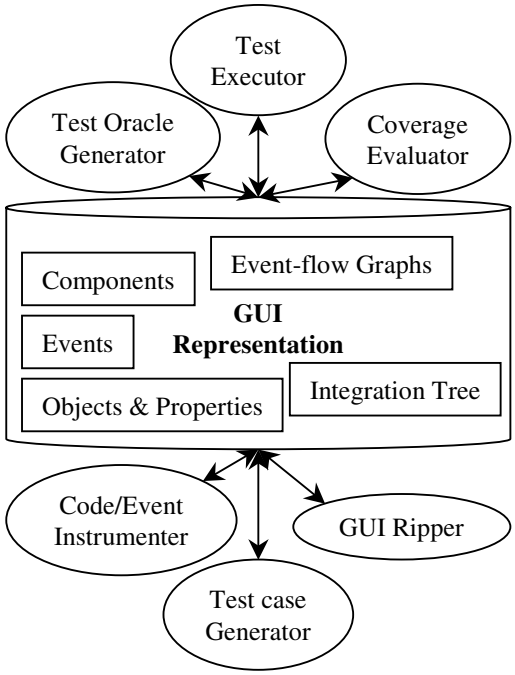

Figure 2. Modules of DART.

4.1. Objects and 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.

4.2. 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 $e_1; e_2; e_3; \ldots; e_n$ 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).

4.3. 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, that 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 that are also examined in the GUI are called *modeless windows*[††] 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*[‡‡].

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.

[**] See http://www.acm.org/sigchi/bulletin/1998.2/students.html.

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

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

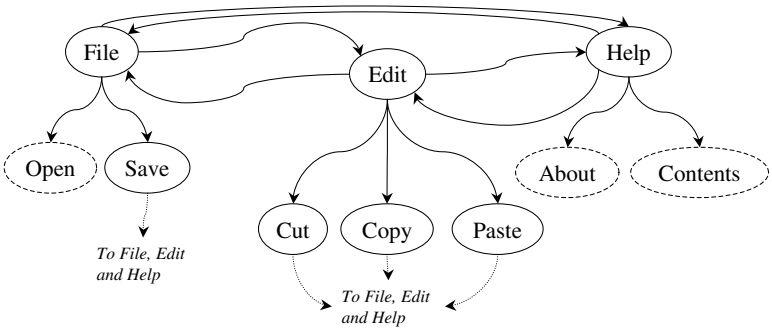

Figure 3. EFG for part of MS WordPad.

4.4. 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 $\langle \mathbf{V}, \mathbf{E}, \mathbf{B}, \mathbf{I} \rangle$ where:

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

Note that an event-flow graph is not a state machine. The nodes represent events in the component and the edges show the `follows` relationship. An example of an event-flow graph for a part of the `Main`* component of MS WordPad is shown in Figure 3. 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., $\mathbf{I} = \{$`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.

*The component that is presented to the user when the GUI is first invoked.

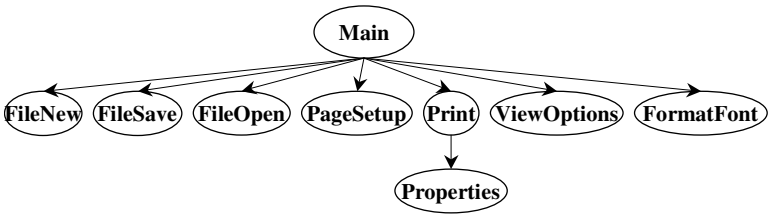

Figure 4. IT for part of MS WordPad.

4.5. 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 follows.

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

Intuitively, the IT shows the invokes relationship among all the components in a GUI. Formally, an IT is defined as follows.

Definition. An *IT* is a triple $\langle \mathcal{N}, \mathcal{R}, \mathcal{B} \rangle$, 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.

Note that, in general, the relationship among components may be represented by a DAG (directed acyclic graph), since multiple components may invoke a component. However, the DAG can be converted into a tree by copying nodes. The tree model simplifies our algorithms based on tree traversals of the IT. Figure 4 shows an example of an IT representing a part of 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 4 has an edge from `Main` to `FileOpen` showing that `Main` contains an event, namely `Open` (see Figure 3), which invokes `FileOpen`.

4.6. Event classification

Creation of the event-flow graphs and IT requires the identification of specific types of events. The classification of GUI events is as follows.

Restricted-focus events open *modal windows*. `Set Language` in Figure 5 is a restricted-focus event.

Unrestricted-focus events open *modeless windows*. For example, `Replace` in Figure 6 is an unrestricted-focus event.

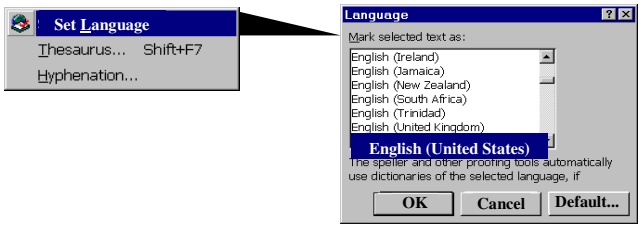

Figure 5. The event `Set Language` opens a modal window.

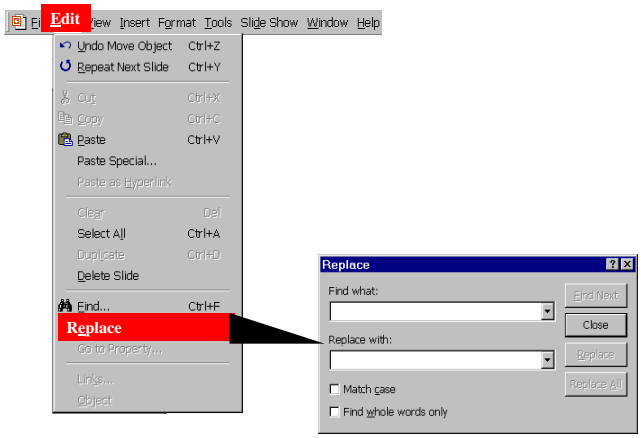

Figure 6. The event `Replace` opens a modeless window.

Termination events close modal windows; common examples include `ok` and `cancel` (Figure 5).

The GUI contains other types of events that do not open or close windows but make other GUI events available. These events are used to open menus that contain several events.

Menu-open events are used to open menus. They expand the set of GUI events available to the user. Menu-open events do not interact with the underlying software. Note that the only difference between menu-open events and unrestricted-focus events is that the latter open windows that must be explicitly terminated. The most common example of menu-open events are generated by buttons that open pull-down menus. For example, in Figure 7, `File` and `SendTo` are menu-open events.

Finally, the remaining events in the GUI are used to interact with the underlying software.

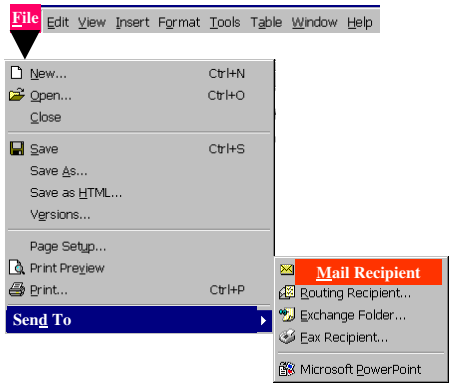

Figure 7. Menu-open events: `File` and `Send To`.

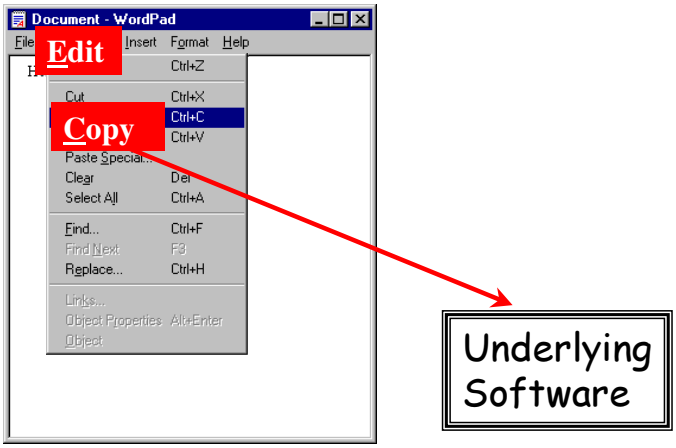

Figure 8. A system-interaction event: `Copy`.

System-interaction events interact with the underlying software to perform some action; common examples include the `Copy` event used for copying objects to the clipboard (see Figure 8).

Table IV lists some of the components of WordPad. Each row represents a component and each column shows the different types of events available within each component. `Main` is the component that is available when WordPad is invoked. Other components' names indicate their functionality. For example, `FileOpen` is the component of WordPad used to open files.

Table IV. Types of events in some components of MS WordPad.

	Event type					
Component name	Menu open	System interaction	Restricted focus	Unrestricted focus	Termination	Sum
Main	7	27	19	2	1	56
FileOpen	0	8	0	0	2	10
FileSave	0	8	0	0	2	10
Print	0	9	1	0	2	12
Properties	0	11	0	0	2	13
PageSetup	0	8	1	0	2	11
FormatFont	0	7	0	0	2	9
Sum	7	78	21	2	13	121

5. MODULES OF DART

Having presented a formal model of the GUI, we now describe each module shown in Figure 2.

5.1. GUI ripper

The GUI ripper is the first module used by the tester/developer to start the smoke testing process. The GUI ripper invokes a process called *GUI ripping* to automatically obtain the GUI's representation. GUI ripping is a dynamic process that is applied to an executing software's GUI. Starting from the software's first window (or set of windows), the GUI is 'traversed' by opening all child windows. All the window's *widgets* (building blocks of the GUI, e.g., buttons, text-boxes), their *properties* (e.g., background-color, font), and *values* (e.g., red, Times New Roman, 18pt) are extracted. Developing this process has several challenges that required us to develop novel solutions. First, the source code of the software may not always be available; we had to develop techniques to extract information from the executable files. Second, there are no GUI standards across different platforms and implementations; we had to extract all the information via low-level implementation-dependent system calls, which we have found are not always well-documented. Third, some implementations may provide less information than necessary to perform automated testing; we had to rely on heuristics and human intervention to determine missing parts. Finally, the presence of *infeasible paths* in GUIs prevents full automation. For example, some windows may be available only after a valid password has been provided. Since the GUI ripper may not have access to the password, it may not be able to extract information from such windows. We had to provide another process and tool support to visually add parts to the extracted GUI model.

The process of GUI ripping consists of two steps. First, the GUI of the application is automatically traversed and its structure is extracted. Second, since the implementation may be wrong (after all, that is what is being tested), the extracted information may be incorrect; the tester visually inspects the extracted GUI structure and makes corrections so that the structures conform to software specifications.

```
PROCEDURE DFS-Trees(DFS-Forest F)
  R /* Set of all root nodes in the forest F */                    1
  FORALL root ∈ R DO                                               2
    DFS-Tree-Recursive(root)                                       3
PROCEDURE DFS-Tree-Recursive(Node n)
  W = get-child-nodes(n)                                           4
  W /* Set of child nodes of the node being visited */             5
  FORALL w ∈ W DO                                                  6
      DFS-Tree-Recursive(w)                                        7
```

Figure 9. Visiting each node in a forest of directed trees.

We first describe the algorithm used for the GUI ripper and then discuss the role of the human tester in inspecting and correcting the extracted structure. We will use a top-down approach to describe our ripping algorithm. Since we use a depth-first search (DFS) traversal of the GUI to extract its structure, we will start with a generalized DFS algorithm and tailor it for GUIs. We have instantiated the same algorithm for Windows and Java implementations [33].

5.1.1. GUI traversal and extraction algorithm

As discussed earlier in Section 4, the GUI of an application is modeled using event-flow graphs and an integration tree. The GUI ripper uses an intermediate representation to create these models. This representation is called a *GUI forest*. Intuitively, a GUI forest represents all the windows in the GUI and the invokes relationship between them. In most simple GUIs, the forest is a single tree. However, since a GUI can have multiple windows (called its top-level windows) when it is first invoked, a forest structure is a more convenient representation. We obtain this structure by performing a DFS traversal of the hierarchical structure of the GUI. We start with a generalized DFS algorithm [39] shown in Figure 9 and adapt it for GUIs.

The procedure `DFS-Trees` takes as its input a forest, represented as a set of trees. It performs a DFS traversal starting from the root of each tree (lines 2–3). The procedure `DFS-Tree-Recursive` visits the tree rooted at node n. A list $\mathcal{W}$ of all the child nodes of the node n is obtained (line 4). Then a recursive visit for the sub-trees rooted at each of the child nodes is performed (line 6–7).

We tailor the algorithm of Figure 9 to handle GUI traversal. The resulting algorithm is shown in Figure 10. Two procedures `DFS-GUI` and `DFS-GUI-Recursive` traverse the GUI of the application and extract its structure. The function `access-top-level-windows` (line 1) returns the list of top-level windows in the AUT. Recall that top-level windows of an application are those windows that become visible when the application is first launched. A GUI tree is constructed for each of the top-level windows by invoking the procedure `DFS-GUI-Recursive`. The trees are constructed in the set $\mathcal{GUI}$. At the termination of the algorithm, $\mathcal{GUI}$ contains the GUI forest of the application.

```
GUI /* GUI tree of application under test */
PROCEDURE DFS-GUI(Application A)
  T = access-top-level-windows(A)                          1
  GUI = T                                                  2
  /* T is set of top-level windows in the application */
  FORALL t ∈ T DO                                          3
    DFS-GUI-Recursive(t)                                   4
PROCEDURE DFS-GUI-Recursive(Window g)
  W = get-widget-list-and-properties(g)                    5
  /* W is the set of all widgets in the Window */
  E = identify-executable-widgets (W)                      6
  /* From W identify executable widgets */
  FORALL e ∈ E DO                                          7
    execute-widget(e)                                      8
    /* Execute the widget e */
    C = get-invoked-gui-windows(e)                         9
    GUI = GUI ∪ g                                          10
    FORALL c ∈ C DO                                        11
      DFS-GUI-Recursive(c)                                 12
```

Figure 10. GUI traversing and extracting the GUI representation of an application.

Note that lines 4–7 of Figure 9 have been replaced with lines 5–12 in Figure 10. This is because, for a directed tree, the children of a node can be obtained by invoking the procedure `get-child-nodes`. However, for a GUI application, a node is a GUI window. It may contain several widgets, which in turn may invoke one or more GUI windows. To obtain a list of all GUI windows that can be invoked from a GUI window g, we must query each of g's constituent widgets.

The procedure `DFS-GUI-Recursive` performs a DFS of the GUI tree rooted at the GUI window g. In line 5 the call to `get-widget-list-and-properties` returns a list $\mathcal{W}$ of the constituent widgets in the GUI window g. The function `identify-executable-widgets` in line 6 searches the set $\mathcal{W}$ and returns a list of widgets which invoke other GUI windows. This is necessary because not all of the widgets in $\mathcal{W}$ invoke other GUI windows.

A widget e that invokes other GUI windows is executed by `execute-widget` in line 8. When executed, e may invoke one or more GUI windows. The function `get-invoked-gui-windows` in line 9 returns the list of GUI windows invoked by e. Note that each of the GUI windows c in the set $\mathcal{C}$ are child nodes of the node g in the GUI tree. The GUI tree $\mathcal{GUI}$ is updated in line 10. This is done by inserting each GUI Window c from $\mathcal{C}$ as a child node of the GUI window g. Lines 11–12 performs a recursive search of the sub-tree rooted at each of the invoked GUI windows c.

When the procedure `DFS-GUI-Recursive` returns to `DFS-GUI`, the tree rooted at the top-level window t is constructed. At the completion of the procedure `DFS-GUI`, the complete GUI forest of the AUT is available in $\mathcal{GUI}$.

The algorithm described in Figure 10 is general and can be applied to any GUI described in Section 4. In earlier work, we have described how the high-level functions used in the algorithm may be implemented using Windows and Java API [33].

5.1.2. Manual inspection

The automated ripping process is not perfect. Different idiosyncrasies of specific platforms sometimes result in missing windows, widgets, and properties. For example, we cannot distinguish between modal and modeless windows in MS Windows; we cannot extract the structures of the `Print` dialog in Java. Such platform specific differences require human intervention. We provide tools to edit and view the extracted information. We also provide a process called 'spy' with which a test designer can manually interact with the AUT, open the window that was missed by the ripper, and add it to the GUI forest at an appropriate location.

5.1.3. Generating the event-flow graph and IT

During the traversal of the GUI, we also determine the event type (discussed in Section 4) by using low-level system calls. Once this information is available, we create the event-flow graphs and IT relatively easily using the algorithms described in [24].

5.2. Test-case generator

Our concepts of events, objects and properties can be used to formally define a GUI test case as follows.

Definition. A *GUI test case* T is a pair $\langle S_0, e_1; e_2; \ldots ; e_n \rangle$, 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 EFGs and the IT 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 outputs and serve as a GUI test case $\langle S_0, e_1; e_2; \ldots ; e_n \rangle$.

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.

5.3. 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 $\langle S_0, e_1; e_2; \ldots ; e_n \rangle$, 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.

There are several different ways to compute the expected state (oracle information). We now outline three of them.

1. Using *capture/replay tools* is the most commonly used method to obtain the oracle information [40]. Capture/replay tools are semi-automated tools used to record and store a tester's manual interaction with the GUI with the goal of replaying it with different data and observing the software's output. The key idea of using these tools is that testers manually select some widgets and some of their properties that they are interested in storing during a capture session. This partial state is used as oracle information during replay. Any mismatches are reported as possible defects.
2. We have used *formal specifications* in an earlier work [36] to automatically derive oracle information. These specifications are in the form of pre/postconditions for each GUI event.
3. For the smoke tester, we have used a third approach that we call *execution extraction*. During this process, a test case is executed on an existing, presumably correct version of the software and its state is extracted and stored as oracle information. We have employed platform-specific technology such as Java API[†], Windows API[‡], and MSAA[§] to obtain this information.

Depending on the resources available, DART can collect and compare oracle information at the following different levels (LOI) of (decreasing) cost and accuracy[¶]. Detailed comparison between these levels is given in Section 6.

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

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

Widget: LOI4 $= \{(w, p, o), (w = \mathit{active\ Window}), o = \mathit{current\ object}, \forall p = \mathit{properties} \in o\}$, 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.

5.4. Coverage evaluator

Although smoke tests are not meant to be exhaustive, we have found that coverage evaluation serves as a useful guide to additional testing, whether it is done for the next build or for future

[†] See java.sun.com.
[‡] See msdn.microsoft.com/library/default.asp?url=/library/en-us/winprog/winprog/windows_api_reference.asp.
[§] See msdn.microsoft.com/library/default.asp?url=/library/en-us/msaa/msaaccrf_87ja.asp.
[¶] The need for these levels is explained in detail in an earlier reported work [36].

comprehensive testing. Also, our use of the matrix M' 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 coverage and event coverage. 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.

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 the algorithms used to compute the matrix are presented in earlier reported work [37]. We have already shown examples of matrices in Tables II and III.

5.5. Event instrumenter

The coverage evaluator requires that all event sequences that are executed on the GUI be collected. We have developed an event-based instrumenter based on our previous work [33]. We now describe the design of the instrumenter.

Recall that GUIs have widgets such as Buttons, Menus, TextFields, and Labels that are the building blocks of a GUI. Some of these widgets (e.g., Buttons, Menus, and TextFields) allow user interactions whereas other widgets are static (e.g., Labels used to display text). Users interact with the widgets by performing events. For example typing a character or pressing a mouse button.

Each widget that handles user events has *event listeners* attached to it. Event listeners are invoked when events are performed on the widgets. For example, a *Mouse-Over* event listener for a toolbar button may display a tool-tip. In Figure 11(a), *actionPerformed* is a method of the *ActionListener* event listener that handles events on the *Save* menu-item. Note that multiple event listeners can be attached to a widget. For example, a TextField may have a key event listener and a mouse event listener attached to it.

The key idea of our instrumenter is to detect the existing listeners and attach our own listeners. Hence, whenever a user performs an event on a particular widget, our listener gets a message. The choice of event listeners depends on the type of the widget. For example ActionListener is a listener that is attached to widgets such as Buttons and Menus, and ItemListener is attached to Checkboxes.

We have implemented the instrumenter in Java. It is implemented as a separate *Thread* of execution and is activated when the application is invoked. In a Java application, all GUI windows

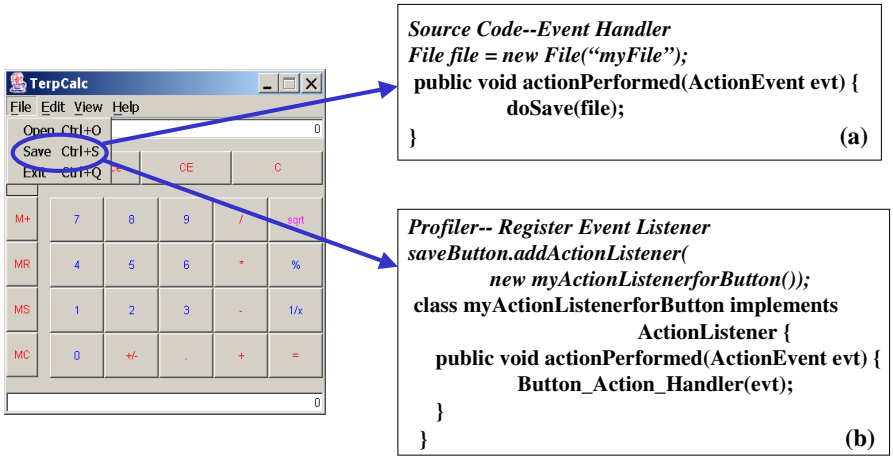

Figure 11. Event-based instrumenter.

and widgets are instances of Java classes. They are analyzed using Java APIs. For example, API *java.awt.Frame.getFrames()* is used to identify all the visible GUI windows of the application. The GUI windows are analyzed, using methods *getComponents* of the class *Container* and *getJMenuBar()* of the class *JFrame*, to extract widgets.

The next step is to analyze the extracted widgets to identify the existing listeners and attach our own listener. For example in Figure 11(b), *myActionListenerforButton()* is the listener that the profiler attaches to the *Save* menu-item at runtime. Hence, whenever a user performs an event or action on *Save*, the profiler gets a message of the event in addition to the default action that the *Save* event performs. The profiler records all of this event information.

5.6. 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 applications and Java API *doClick* for Java applications.

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 ten different such combinations.

Table V. Possibilities available to the test designer for level of detail of oracle information.

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

Table V shows these combinations, marked with an '×'. Note that information cannot be used unless it has been generated, i.e., if only LOI4 is available, then LOT1–LOT3 cannot be performed. We compare these combinations in an experiment in the next section.

6. EXPERIMENTS

Having presented the design of DART, we now examine its practicality using actual test runs, report execution time and space requirements, and the code covered by the smoke test cases.

Open questions

We identified the following 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 space) of generating detailed test oracle information?
3. What is the additional cost of test execution when using detailed test oracle information?
4. What is the code coverage of the smoke test cases?

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.

Subject applications

For our study, we used six Java programs as our subjects. These programs were developed as part of an OpenSource office suite software[‖]. Table VI describes these subjects, showing the number of

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

Table VI. Our subject applications.

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	31 575	92	24

windows, lines of code (LOC), number of classes and number of components. Note that these are not toy programs. In all, they contain more than 30 000 LOC, with at least two programs of almost 10 000 LOC.

6.1. Experimental design

6.1.1. Variables

In the experiment, we manipulated three independent variables as follows.

1. P: the subject programs (six programs).
2. LOI: level of oracle information detail (four levels: complete, complete visible, active window, widget).
3. LOT: levels of testing (four 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 'ripped' the GUI, generated smoke test cases and measured the total generation time and space required. The exact number of test cases that we generated and their lengths are shown for each application in Tables VII–XII. Note that the maximum number of test cases were generated for TerpPaint (Table VII), which has a complex user interface, especially the drawing toolbars. Since we wanted to test all interactions between drawing tools, we generated a large number of length-2 test cases; no length-3 test cases were generated. TerpPad (Table VIII) and TerpSpreadSheet (Table IX) were less complex; we chose to generate approximately 5000 test cases for each. TerpCalc (Table X) has only one window with many buttons; we again chose to test a large number of interactions by generating a large number of length-2 test cases. TerpDraw (Table XI) and TerpManager (Table XII) have simple GUIs, with TerpManager having only one modal window. We generated less than 3000 test cases for each of these applications. We then (code + event) instrumented each application and executed all these test cases for each of the ten possible LOI and LOT combinations (Table V).

Table VII. Number of smoke test cases for TerpPaint.

	Test case length		
Component name	1	2	3
Main	81	6500	0
Open_1	16	225	0
Save_2	19	324	0
Choose a file to import. . . _3	19	324	0
rotate_4	8	56	0
stretch_5	6	30	0
Attribute_6	9	72	0
Choose Background Color_7	28	729	0
Total per length	186	8260	0
Total			8446

Table VIII. Number of smoke test cases for TerpPad.

	Test case length		
Component name	1	2	3
Main	12	148	100
Open_1	18	289	800
Save_2	19	324	800
Save_3	19	324	800
Go To Line_6	3	4	4
Change Font_7	21	400	1000
Encrypte/Decrypt_9	3	4	4
Total per Length	95	1493	3508
Total			5096

6.1.2. Threats to validity

- *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 Java Virtual Machine each time.

Table IX. Number of smoke test cases for TerpSpreadSheet.

	Test case length		
Component name	1	2	3
Main	28	780	500
Open_1	19	324	100
Save_2	19	324	100
Find_3	5	16	20
Format Cells_4	14	156	100
Background Color_5	28	729	500
Font Color_6	28	729	500
Column_Width_7	3	4	4
Row Height_8	3	4	4
Total per length	147	3066	1828
Total			5041

Table X. Number of smoke test cases for TerpCalc.

	Test case length		
Component name	1	2	3
Main	77	5865	0
Total per length	77	5865	0
Total			5942

Table XI. Number of smoke test cases for TerpDraw.

	Test case length		
Component name	1	2	3
Main	8	70	200
Open_1	19	324	1000
Save_2	19	324	1000
Total per length	46	718	2200
Total			2964

Table XII. Number of smoke test cases for TerpManager.

	Test case length		
Component name	1	2	3
Main	27	702	1500
Total per length	27	702	1500
Total			2229

- *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 a multiple number of times and report the average time.

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

6.2. Results

6.2.1. Space requirements

We expected DART to have significant space requirements, since it requires the generation of the GUI representation, test cases, oracle information, and test results. Figure 12 shows the space requirements for the six subject programs. LOI0 represents test cases that contain no oracle information, i.e., the LOI0 column shows the space required to store the GUI representation and test cases. We had expected that the space requirements would increase as the level of oracle detail increases. Figure 12 shows that the space requirements grow very rapidly when using a detailed level of test oracle. Note that we are using a logarithmic scale to improve readability. The space demands are not so serious for our smaller subject programs. However, they become very high for large programs (TerpPaint and TerpDraw) that contain a large number of windows. The space requirements also depend on the number of smoke test cases that we generated. Recall that we generated a large number of test cases for TerpPaint, TerpCalc, and TerpSpreadSheet; they required the maximum disk space. On the other hand, even though a large number of test cases were generated for TerpPad, it required less space because of the small number of widgets and objects in its GUI.

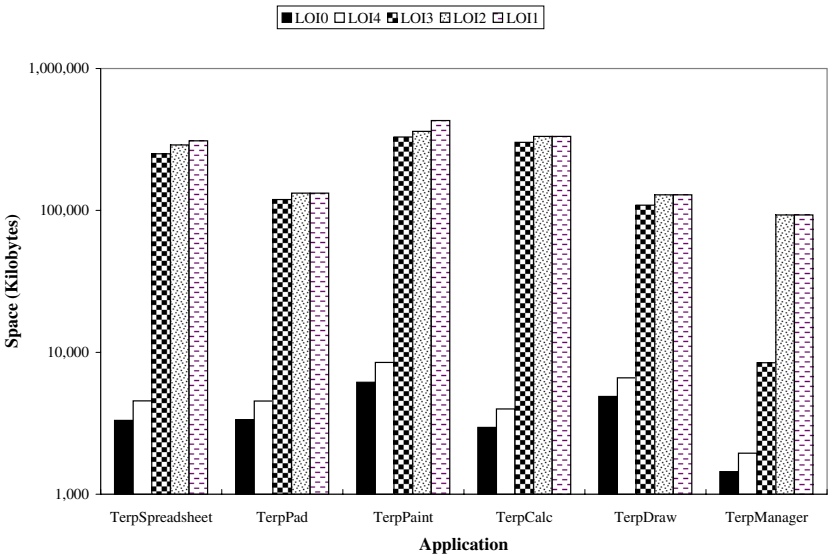

Figure 12. Space requirements of our subject applications for different levels of detail of oracle information.

Even though some of our subject applications required more space than we expected, given the large sizes and low cost of today's hard-disks, we feel that this space requirement (500 MB for TerpPaint) is reasonable. We also note that no attempt was made on our part to save space, i.e., all the files used for the representation were human-readable in XML. We could easily get upto 90% compression if we used a more efficient binary encoding.

6.2.2. Time requirements

The time required for the overall DART process includes `SETUP` that contains application ripping, and test-case, and test-oracle generation, followed by the `ITERATIVE` phase. In earlier work [33], we have demonstrated that ripping and test-case generation are very fast, requiring a few minutes per application. Test oracle generation requires that all test cases be executed and oracle information be collected. The iterative process time is also dominated by the test-case execution time. We therefore measured the test-case execution time for all LOIs and LOTs. All times are reported for a 2.2 GHz Pentium 4 machine with 256 MB of RAM.

The results of this experiment are shown in Figures 13–18. The *y*-axis shows the time in seconds and the *x*-axis shows the LOI. In each figure, we have ten data points, corresponding to the '×' in Table V, grouped into four curves (one for each LOT). For example, the total time taken to execute all 8446 test cases on TerpPaint with LOT4 and LOI = Widget was 147 000 s. The time increased close to 185 000 s for LOT1 and LOI = All Windows. With the exception of TerpPaint, all of our applications could be tested in one night. Examination of TerpPaint revealed that 3–4 s per test case were lost because of

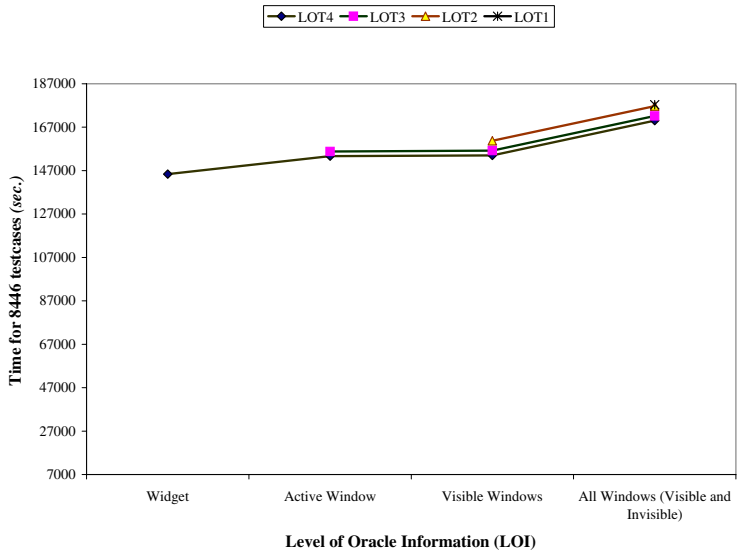

Figure 13. Total execution times for TerpPaint for different LOIs and LOTs.

Figure 14. Total execution times for TerpPad for different LOIs and LOTs.

Figure 15. Total execution times for TerpCalc for different LOIs and LOTs.

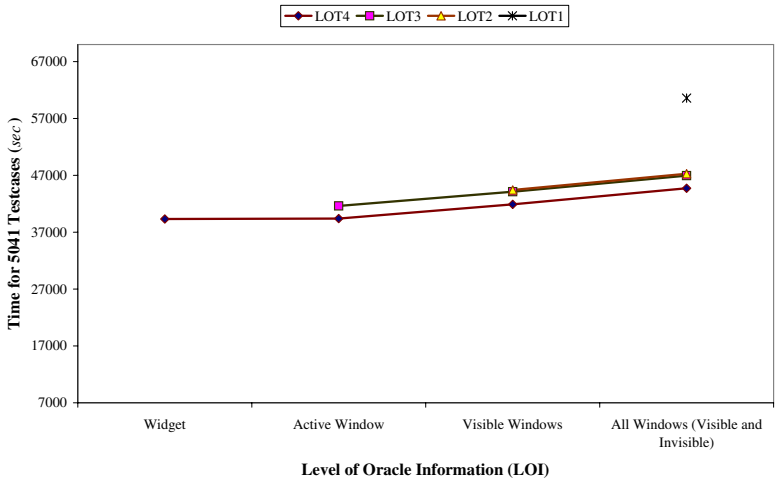

Figure 16. Total execution times for TerpSpreadSheet for different LOIs and LOTs.

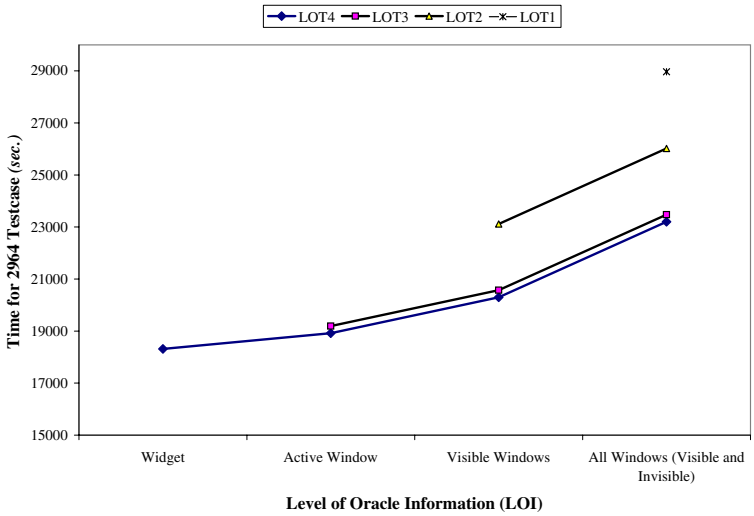

Figure 17. Total execution times for TerpDraw for different LOIs and LOTs.

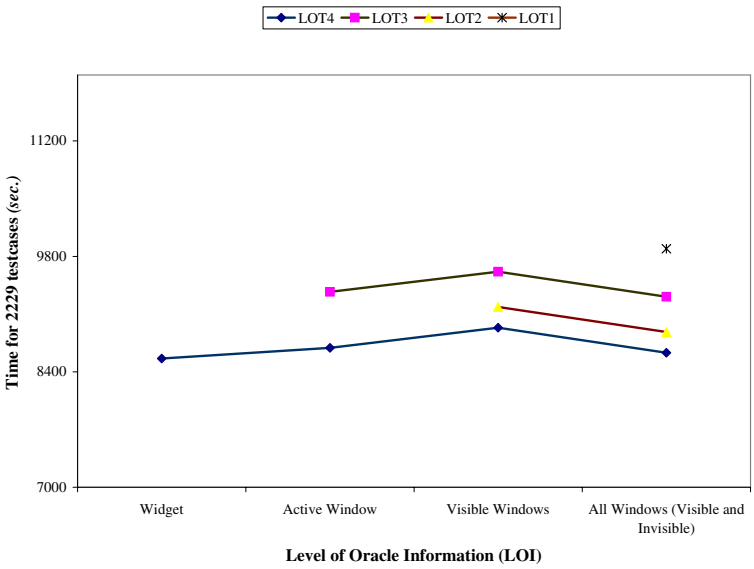

Figure 18. Total execution times for TerpManager for different LOIs and LOTs.

Table XIII. Code coverage for TerpCalc.

	Test case length			
Class name	1	2	3	Total instrumented lines
AboutInfoBox.java	16	16	0	39
BigNumber.java	110	173	0	231
BrowserControl.java	0	0	0	24
Frame2.java	1328	1925	0	3026
Graph.java	51	51	0	286
Grapher.java	85	85	0	347
Gui_Front_End.java	23	23	0	44
Mystring.java	0	2	0	8
StandardFunctions.java	0	0	0	10
StatDat.java	0	6	0	20
StatFrame.java	65	73	0	189
UMClipObj.java	0	0	0	21
backend.java	89	144	0	216
convToPostfix.java	0	0	0	152
Splash.java	0	0	0	55

a splashscreen. We also noted that TerpPad exhibited different behavior with LOI = Widget taking much longer than the other (seemingly more expensive) LOIs. We attribute this result to the additional algorithms implemented to traverse the TerpPad GUI and extract individual widgets.

Our results show that the smoke testing process is practical, in that it can be performed in one night. If developers want to run a large number of longer smoke test cases (i.e., those that require more than one night), we also note that the test cases are completely independent of one another and could easily be distributed across several machines, making the smoke testing process even faster. The coverage and bug reports can be merged to provide feedback to developers.

6.2.3. Code coverage

Since our smoke testing process is iterative, we have found that examining the code coverage of the smoke test cases helps to improve the overall testing process. The tester can focus on missed parts by either generating additional GUI smoke tests that can be run during the next smoke testing cycle or create non-GUI tests using tools such as JUnit. To observe the code coverage of our smoke test cases, we instrumented the applications before running all of our smoke test cases. We recorded the statements that were executed for each user-implemented class during test-case execution. The results of our experiment are shown in Tables XIII–XVIII. The first column in these tables shows names of individual classes, stored in different files. Columns 2, 3 and 4 show the number of statements covered by test cases of length 1, 2 and 3, respectively. The last column shows the total number of statements instrumented. Note that statements do not correspond to lines in the source files; these are source statements as identified by a compiler's syntax analyzer.

Table XIV. Code coverage for TerpPaint.

	Test case length			
Class name	1	2	3	Total instrumented lines
AutoBackup.java	7	7	0	13
Faint.java	1678	1937	0	2963
FaintContainer.java	6	6	0	14
FaintFileFilter.java	38	38	0	60
Fill.java	0	0	0	37
HTMLDisplay.java	0	0	0	13
HelpWindow.java	0	0	0	183
ImageUtilities.java	0	0	0	15
ScrollablePicture.java	7	7	0	26
SlideShow.java	0	6	0	119
Text.java	0	0	0	117
UMClipObj.java	0	0	0	25
about.java	0	0	0	3
attributes.java	187	240	0	254
brushTool.java	6	6	0	126
bucketTool.java	2	2	0	31
canvas_size.java	0	111	0	223
converter.java	0	0	0	309
curveTool.java	3	7	0	116
ellipseTool.java	0	2	0	54
eraserTool.java	3	3	0	55
letterTool.java	3	3	0	10
lineTool.java	3	5	0	37
main_canvas.java	83	110	0	175
medicineTool.java	2	2	0	15
pencilTool.java	2	2	0	46
polyconTool.java	2	6	0	81
printer.java	2	2	0	69
rectTool.java	0	2	0	72
rotate.java	125	145	0	162
roundedRectTool.java	0	2	0	69
saveChanges.java	0	0	0	74
selectTool.java	3	10	0	270
selectallTool.java	3	29	0	272
splash.java	0	0	0	55
sprayTool.java	4	4	0	43
stretch.java	138	138	0	168
viewBitmap.java	0	16	0	25
zoomTool.java	3	3	0	18

Table XV. Code coverage for TerpDraw.

	Test case length			
Class name	1	2	3	Total instrumented lines
DrawingArea.java	155	268	268	2030
OODFilter.java	5	5	5	8
ObjectOrientedDrawer.java	20	24	24	32
Toolbar.java	71	71	71	128
coverter.java	0	0	0	309

Table XVI. Code coverage for TerpPad.

	Test case length			
Class name	1	2	3	Total instrumented lines
ChangeFontDialog.java	157	157	157	188
EncryptDialog.java	30	30	30	34
FileManager.java	0	0	0	77
FindDialog.java	0	49	49	54
FontMacros.java	91	91	91	187
GoToDialog.java	34	34	34	36
HTMLDisplay.java	0	0	0	22
OpenList.java	50	50	50	75
PageCounter.java	10	12	10	12
RC4.java	21	21	21	33
ReplaceDialog.java	0	48	48	58
SplashWindow.java	0	0	0	29
StatusBar.java	18	43	18	92
TwoContentArea.java	135	156	202	532
TwoMenus.java	327	346	365	477
TwoPad.java	54	54	54	109
TwoStyledDocument.java	2	2	2	40
WordCount.java	0	66	66	74

From the tables, we noted that some classes were not at all covered by the smoke tests. We identified two reasons for this result. First, our smoke tests were inadequate, which we easily fixed by specifying additional test cases in the matrix M'. Second, we noted that some parts of the code could never be executed by the smoke test cases because it required setting up execution context using event sequences longer than three. Developers will need to add specific test cases to cover these parts of the code. We did, however, observe that all the GUI-related classes, i.e., Frame2.java (Table XIII), Faint.java (Table XIV), DrawingArea.java (Table XV), TwoMenus.java (Table XVI), ChangeFontDialog.java (Table XVI), FormatMenuOp.java (Table XVII), TerpOffice.java

Table XVII. Code coverage for TerpSpreadSheet.

	Test case length			
Class name	1	2	3	Total instrumented lines
CellPoint.java	7	15	17	35
CellRange.java	17	24	33	96
ClearUndo.java	0	0	13	23
Config.java	32	32	32	77
CopyPaste.java	0	0	20	27
CutUndo.java	12	12	12	22
Debug.java	4	6	6	7
EditMenuOp.java	82	137	146	250
FileHistory.java	21	21	21	59
FileMenuOp.java	42	54	54	196
FindDialog.java	24	24	24	28
FormatCellsTabs.java	150	152	154	173
FormatMenuOp.java	233	272	293	566
Formula.java	11	137	137	415
Function.java	0	21	21	127
FunctionsMenuOp.java	4	109	109	178
GraphsMenuOp.java	4	17	17	119
HTMLDisplay.java	0	0	0	16
HelpMenuOp.java	3	3	3	8
LineBorder.java	41	64	64	95
Node.java	13	42	42	120
NumberField.java	21	21	21	57
ParserException.java	3	5	5	12
PasteUndo.java	0	23	23	39
SharpDialog.java	37	37	37	92
Splash.java	0	0	0	31
UMCell.java	69	76	76	169
UMCellEditor.java	5	5	5	50
UMCellRenderer.java	55	55	55	60
UMClipObj.java	11	16	16	23
UMDialog.java	40	42	42	92
UMOptionPane.java	33	31	33	63
UMSSpreadSheet.java	476	506	514	660
UMTableModel.java	210	334	424	752

(Table XVIII), and TerpOfficeExplorer.java (Table XVIII) were covered by the smoke test cases. Note that these GUI-related classes are difficult to cover using other code-based test cases since they require creating instances of GUI widgets and executing events on them.

From this result, we observed that the coverage reports from the smoke tests were useful to guide the overall smoke testing process. We also discovered that DART and conventional tools such as JUnit have complementary strengths: DART is better-suited for GUI code whereas JUnit is better for testing the underlying 'business logic' code.

Table XVIII. Code coverage for TerpManager.

	Test case length			
Class name	1	2	3	Total instrumented lines
ClipboardViewer.java	51	51	51	64
ImageFileView.java	22	22	22	34
IntegratorClipboardObject.java	4	17	17	78
TerpManagerSplash.java	0	0	0	26
TerpOffice.java	179	179	188	324
TerpOfficeExplorer.java	271	298	311	656
TerpOfficeFileFilter.java	37	37	37	60
fileProperties.java	0	0	0	177

7. RELATED WORK

Although there is no prior work that directly addresses the research presented in this paper, several researchers and practitioners have discussed concepts that are relevant to its specific parts. We discuss the following broad categories: *daily builds*, *tool support*, *eXtreme programming*, and *GUI testing tools*.

Daily builds

Daily building and smoke testing has been used for a number of large-scale projects, both commercial and OpenSource. For example, Microsoft used daily builds extensively for the development of its popular Windows NT operating system [3]. 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 its success on that huge project to their daily builds.

The GNU project (http://www.gnu.org) continues to use daily builds for most of its projects. For example, during the development of Ghostscript software daily builds were used widely. The steps in the daily builds involved preparing the source code by compiling the source, executing the smoke tests and updating the CVS and making the source archive. Similarly, *WINE* [7], *Mozilla* [8], *AceDB* [8] and *openwebmail* [10] use nightly/daily builds.

Tool support

There are a number of tools available that help when performing the smoke testing of software applications. For example, *CruiseControl* [16] is a framework for a continuous build process. It includes, plug-ins for e-mail notification, Ant, and various source control tools. A Web interface is provided to view the details of the current and previous builds. The continuous build process allows each developer to integrate daily, thus reducing integration problems. *IncrediBuild* [17] is a tool that speeds up compilation by performing distributed compilation of source by distributing the compilation task across 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. Similarly, other tools such as Daily Build [18] and *Visual Build* [19] support daily builds. While there are many projects that use daily builds, there is no literature on techniques and tools for daily builds and smoke tests of GUI software.

eXtreme programming

A closely related paper discusses automating acceptance tests for GUIs in an extreme programming environment [29] 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 [29,41]. 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 [29,42]; 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 [43] frameworks, such as jUnit and jfcUnit. GUI widgets are accessed from the GUI and tested for existence and functionality [44]. 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.

GUI testing tools

There has been much more in the automation of GUI testing. Most of the techniques use Capture/Replay tools for testing GUIs. These tools operate in two modes *Record* and *Playback* mode. In the *Record* mode, tools such as *CAPBAK* and *TestWorks* [45] 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 test cases might break even with the slightest change in the layout of the GUI. Tools such as *Winrunner* [46], *Abbot* [47], and *Rational Robot* [48] enable capturing GUI widgets rather than coordinates thereby solving the problem. The testing technique followed by *Rational Robot* for the GUIs of Web applications allows: recording and replaying test scripts that recognize the objects in various applications; tracking and reporting information about the quality assurance testing process; detection and repairing problems in the elements of a Web site; and viewing and editing test scripts. Although it allows automation, significant effort is involved in creating test scripts, detecting failures, and editing the tests to make it work on the modified version of software. Even with these capabilities, these tools do not support the smoke testing of GUI software. Our DART framework enables the smoke testing of GUIs by automatic test-case generation and automated test oracles to determine failures.

8. CONCLUSIONS AND FUTURE WORK

Today's large software systems are often maintained by several programmers, who may be geographically distributed. To ensure quality of these systems, nightly builds and smoke tests have become widespread as they often reveal bugs early in the software maintenance process. Although successful for conventional software, smoke tests are difficult to develop and automatically re-run for

software that has a GUI. In this paper, we have presented a technique for smoke 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. We described the primary modules of DART that automate the entire smoke testing process. Even though we present DART as a smoke testing tool, it is efficient enough to be used for any type of frequent GUI re-testing. We also note that the GUI smoke tests are not meant to replace other code-based smoke tests. DART is a valuable tool to add to the tool-box of the tester/developer.

In the future, we will study the effectiveness of the DART process by analyzing the number of faults detected. We will also integrate DART in a higher-level process that involves executing other types (non-GUI) of smoke tests on the software. We will also investigate the application of DART to other software systems that take events as input. One example of such systems are Web applications.

REFERENCES

1. Memon AM, Banerjee I, Hashmi N, Nagarajan A. DART: A framework for regression testing nightly/daily builds of GUI applications. *Proceedings International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 2003; 410–419.
2. Karlsson E-A, Andersson L-G, Leion P. Daily build and feature development in large distributed projects. *Proceedings of the 22nd International Conference on Software Engineering*. ACM Press: New York NY 2000; 649–658.
3. McConnell S. Best practices: Daily build and smoke test. *IEEE Software* 1996; **13**(4):143–144.
4. Olsson K. Daily build—the best of both worlds: Rapid development and control. *Technical Report*, Swedish Engineering Industries, 1999.
5. Robbins J. *Debugging Applications*. Microsoft Press, 2000.
6. Halloran TJ, Scherlis WL. High quality and open source software practices. *Meeting Challenges and Surviving Success: Proceedings of the 2nd Workshop on Open Source Software Engineering*. ACM Press: New York NY, 2002.
7. WINE Daily Builds. http://wine.dataparty.no/ [2003].
8. Mozilla. http://ftp.mozilla.org/pub/mozilla/nightly/latest/ [2003].
9. Current Daily Builds of AceDB. http://www.acedb.org/Software/Downloads/daily.shtml [2003].
10. Open WebMail. http://openwebmail.org/openwebmail/download/redhat/rpm/daily-build/ [2003].
11. Marick B. When should a test be automated? *Proceedings of the 11th International Software/Internet Quality Week*. Software Research Institute: San Francisco CA, 1998.
12. Crispin L, House T, Wade C. The need for speed: Automating acceptance testing in an eXtreme Programming environment. *Proceedings of the Second International Conference on eXtreme Programming and Flexible Processes in Software Engineering*. Addison-Wesley: Boston MA, 2001; 96–104.
13. Grenning J. Launching eXtreme programming at a process intensive company. *IEEE Software* 2001; **18**:27–33.
14. Schuh P. Recovery, redemption and extreme programming. *IEEE Software* 2001; **18**:34–41.
15. Poole C, Huisman JW. Using extreme programming in a maintenance environment. *IEEE Software* 2001; **18**:42–50.
16. Cruise Control. http://cruisecontrol.sourceforge.net/ [2003].
17. FAST C++ Compilation—IcrediBuild by Xoreax Software. http://www.xoreax.com/main.htm [2003].
18. Positive-g- Daily Build Product Information—Mozilla. http://positive-g.com/dailybuild/ [2003].
19. Kinook Software—Automate Software Builds with Visual Build Pro. http://www.visualbuild.com/ [2003].
20. Salzman MC, Rivers SD. Smoke and mirrors: Setting the stage for a successful usability test. *Behaviour and Information Technology* 1994; **13**(1/2):9–16.
21. Rothermel G, Harrold MJ. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology* 1997; **6**(2):173–210.
22. Binkley D. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering* 1997; **23**(8):498–516.
23. Rosenblum DS, Weyuker EJ. Using coverage information to predict the cost-effectiveness of regression testing strategies. *IEEE Transactions on Software Engineering* 1997; **23**(3):146–156.
24. Memon AM. A comprehensive framework for testing graphical user interfaces. *Doctoral Dissertation*, Department of Computer Science, University of Pittsburgh, July 2001.

25. Memon AM. GUI testing: Pitfalls and process. *IEEE Computer* 2002; **35**(8):90–91.
26. Memon AM, Soffa ML. Regression testing of GUIs. *Proceedings of the 9th European Software Engineering Conference (ESEC) and 11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-11)*. ACM Press: New York NY, 2003, 118–127.
27. Myers BA. User interface software tools. *ACM Transactions on Computer–Human Interaction* 1995; **2**(1):64–103.
28. Marick B. Bypassing the GUI. *Software Testing and Quality Engineering Magazine* 2002; (September):41–47.
29. Finsterwalder M. Automating acceptance tests for GUI applications in an eXtreme programming environment. *Proceedings of the 2nd International Conference on eXtreme Programming and Flexible Processes in Software Engineering*. Addison-Wesley: Boston MA, 2001; 114–117.
30. White HAL, Alzeidi N. User-based testing of GUI sequences and their interactions. *Proceedings 12th International Symposium Software Reliability Engineering*. IEEE Computer Society Press: Los Alamitos CA, 2001; 54–63.
31. Hicinbothom JH, Zachary WW. A tool for automatically generating transcripts of human–computer interaction. *Proceedings of the Human Factors and Ergonomics Society 37th Annual Meeting, vol. 2 Special Sessions: Demonstrations*. Human Factors and Ergonomics Society: Santa Monica CA, 1993; 1042.
32. Memon AM. Advances in GUI testing. *Advances in Computers*, vol. 57, Zelkowitz MV (ed.). Academic Press: New York NY, 2003.
33. Memon AM, Banerjee I, Nagarajan A. GUI ripping: Reverse engineering of graphical user interfaces for testing. *Proceedings 10th Working Conference on Reverse Engineering*. IEEE Computer Society Press: Los Alamitos CA, 2003.
34. Memon AM, Pollack ME, Soffa ML. Hierarchical GUI test case generation using automated planning. *IEEE Transactions on Software Engineering* 2001; **27**(2):144–155.
35. Memon AM, Pollack ME, Soffa ML. Using a goal-driven approach to generate test cases for GUIs. *Proceedings of the 21st International Conference on Software Engineering*. ACM Press: New York NY, 1999; 257–266.
36. Memon AM, Pollack ME, Soffa ML. Automated test oracles for GUIs. *Proceedings of the ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering (FSE-8)*. ACM Press: New York NY, 2000; 30–39.
37. Memon AM, Soffa ML, Pollack ME. Coverage criteria for GUI testing. *Proceedings of the 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)*. ACM Press: New York NY, 2001; 256–267.
38. Java Source Code Instrumentation. http://www.glenmccl.com/instr/instr.htm [2003].
39. Cormen T, Leiserson C, Rivest R. *Introduction to Algorithms*, ch. 23.3. MIT Press: Cambridge MA, 1990; 477–485.
40. Kepple LR. The black art of GUI testing. *Dr Dobb's Journal of Software Tools* 1994; **19**(2):40.
41. Beck K. *eXtreme Programming Explained: Embrace Change*. Addison-Wesley: Reading MA, 1999.
42. Kaner C. Improving the maintainability of automated test suites. *Proceedings of the 10th International Software/Internet Quality Week*, 1997.
43. eXtreme Programming. http://www.XProgramming.com/software.htm [2003].
44. Jeffries R, Anderson A, Hendrickson C. *eXtreme Programming Installed*. Addison Wesley: Reading MA, 2001.
45. Capture–Replay Tool. http://soft.com [2003].
46. Mercury Interactive WinRunner. http://www.mercuryinteractive.com/products/winrunner [2003].
47. Abbot Java GUI Test Framework. http://abbot.sourceforge.net [2003].
48. Rational Robot. http://www.rational.com.ar/tools/robot.html [2003].

AUTHORS' BIOGRAPHIES

Atif M. Memon is an Assistant Professor at the Department of Computer Science, University of Maryland. He received his BS, MS, and PhD in Computer Science in 1991, 1995, and 2001, respectively. He was awarded a Gold Medal in BS. He was awarded Fellowships from the Andrew Mellon Foundation for his PhD research. 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.

Adithya Nagarajan received his BE from the Visvesvaraya Regional College of Engineering, Nagpur, India in 2000. He was awarded a Gold Medal during his BE. He completed his MS in Systems Engineering from the University of Maryland, College Park in 2003. His research interests include software engineering, software quality, software maintenance and reverse engineering. He is a student member of the ACM and the IEEE Computer Society.

Qing Xie is a PhD student at the University of Maryland, College Park. Her research interests include GUI testing, object-oriented testing, and mutation testing.