Rapid "Crash Testing" for Continuously Evolving GUI-Based Software Applications

Qing Xie and Atif M. Memon
Department of Computer Science, University of Maryland
{qing, atif}@cs.umd.edu

Abstract

Several rapid-feedback-based quality assurance mechanisms are used to manage the quality of continuously evolving software. Even though graphical user interfaces (GUIs) are one of the most important parts of software, there are currently no mechanisms to quickly retest evolving GUI software. We leverage our previous work on GUI testing to define a new automatic GUI re-testing process called "crash testing" that is integrated with GUI evolution. We describe two levels of crash testing: (1) immediate feedback-based in which a developer indicates that a GUI bug was fixed in response to a previously reported crash; only select crash test cases are rerun and the developer is notified of the results in a matter of seconds, and (2) between code changes in which new crash test cases are generated on-the-fly and executed on the GUI. Since the code may be changed by another developer before all the crash tests have been executed, hence requiring restarting of the process, we use a simple rotation-based scheme to ensure that all crash tests are executed over a series of code changes. We show, via empirical studies, that our crash tests are effective at revealing serious problems in the GUI.

1 Introduction

Today's competitive software market trends are increasingly pushing for globalization of development, maintenance, and evolution [32, 28]. Software companies use the world's different time-zones to their advantage by distributing their development teams around the world, thereby enabling continuous around-the-clock software evolution that "follows the sun" [16]. Open-source software, typically developed and maintained by a number of programmers distributed world-wide, is also subject to similar continuous evolution trends. This practice of around-the-clock evolution has led to unprecedented code churn rates. For example, the open-source ACE+TAO software [1] developers average 200+ CVS commits per week [22].

While successful at increasing code churn rates, continuous software evolution suffers from several problems. First, there is little direct inter-developer communication [31]. Almost all communication is done via CVS commit log messages, bug reports, change-requests, comments [29, 9], etc. Second, developers often work on different parts of the application code [31]. They may not immediately realize that their changes have inadvertently broken other parts of the software code [21]. If left unsolved, these problems lead to compromised software quality.

There are several feedback-based mechanisms to help manage the quality of continuously evolving software. These different mechanisms improve the quality of software via continuous, rapid quality assurance (QA) during evolution. They differ in the level of detail of feedback that they provide to the developer, their thoroughness, their frequency of execution, and their speed of execution. For example, some mechanisms (*e.g.*, integrated with CVS) provide immediate feedback at change-commit time by running select test cases, which form the *commit validation suite*. Developers can immediately see the consequences of their changes. For example, developers of NetBeans perform several quick validation steps when checking into the NetBeans CVS repository.[1] In fact, some systems such as Aegis[2] will not allow a developer to commit changes unless all commit-validation tests have passed. This mechanism ensures that changes will not stop the software from "working" when they are integrated into the software baseline. Other, slower mechanisms include "daily building and smoke testing" that execute more thorough test cases on a regular (*e.g.*, nightly) basis. Developers don't get instant feedback; rather they are e-mailed the results of the nightly builds and smoke tests. Another, still higher level of continuous quality assurance support is provided by mechanisms such as Skoll [22] that continuously run test cases, for days and even weeks on several builds (stable and beta) of the evolving software. For example, the ACE+TAO software is tested continuously by Skoll; results are summarized in a web-based virtual scoreboard.[3] All these mechanisms are useful, in that they help to detect defects early during software evolution, much before the software is given to QA teams.

Graphical-user interfaces (GUIs) are one of the most important parts of software. We have shown in earlier work that, due to the number of degrees of freedom that GUIs provide to users, and hence their large input spaces, GUIs

[1] http://www.netbeans.org/community/guidelines/commit.html
[2] http://aegis.sourceforge.net/
[3] http://www.dre.vanderbilt.edu/scoreboard/

are difficult to test [25, 21]. A tester needs to test the GUI on an enormous number of event sequences. Moreover, as GUI-based software evolves, *i.e.*, changes are made to the GUI and the underlying software, it is easy to break [26].

In previous work, we have addressed the GUI testing problem at several levels. First, we have developed specifications-based testing techniques that use a formal model of the GUI's events, encoded as preconditions and effects to generate test cases [25] and test oracles [24]. While the specifications-building process is resource intensive, the generated test cases are effective, and the error reporting is detailed, accurate, and useful. Second, we have developed a faster daily-building and smoke-testing process for GUIs [21]. No manually-created specifications are needed; the correctness of the latest GUI version is checked against the previous (baseline) version. We have found the daily smoke testing process to be useful, in that it provides feedback to developers in a reasonable amount of time, *i.e.*, daily. Although this process is much faster and more automated than our specifications-based approach, there are several problems that need manual intervention. For example, our smoke test cases report some false positives, *i.e.*, some of the bugs that they report are not real bugs – they are simply a consequence of changes made to the GUI during evolution. Moreover, some of our test cases lead to software crashes. These crashes sometimes kill our smoke testing process, causing substantial delays.

We now define a tighter GUI testing cycle that is fully automatic. The key idea is to test the GUI each time it is modified, *e.g.*, at each code commit. Our aim is not to exhaustively test the GUI; rather, we want to quickly raise a "something is wrong here" alarm by checking that each GUI event and interactions between them work correctly. We eliminate the need for manual intervention by requiring neither specifications nor the identification of false positives. We have defined a simple, yet effective, method to determine the success of a test case. We say that the GUI fails on a test case if it causes the software to throw an uncaught exception; otherwise it passes. Consequently, we call this process "crash testing." We describe two levels of crash testing: (1) immediate feedback-based in which a developer indicates that a GUI bug was fixed in response to a previously reported crash; only the select crash test cases that caused the earlier crash are rerun and the developer is notified of the results in a matter of seconds, and (2) between code changes in which new crash test cases are generated on-the-fly and executed on the GUI. Since the code may be changed by another developer before all the crash tests have been executed, hence requiring restarting of the process, we use a simple rotation-based scheme to ensure that all crash tests are executed over a series of code changes.

To generate crash tests, we define a new representation of the GUI called the *event-interaction graph*. This representation models specialized user interactions with the GUI and can be obtained automatically using our reverse engineering techniques [23]. We annotate this graph by adding boolean flags on edges allowing us to develop a rotation-based scheme for generating crash tests. We present results of empirical studies in which we generate crash tests and study their characteristics. In particular, we study the tradeoffs between the number of crash tests and the number of crashes reported. We artificially control the time interval between GUI code changes and study how rotating crash tests helps to quickly eliminate those software defects that manifest themselves as software crashes.

The specific contributions of this work include:

1. Definition of new type of test cases for GUIs that can be generated automatically and executed quickly.
2. Introduction of annotated event-interaction graphs and their application.
3. Empirical studies demonstrating (1) the tradeoffs between the number of crash tests and number of crashes reported and (2) how the simple rotation algorithm helps to speed-up defect detection.

Structure of the paper: In the next section, we identify the requirements of GUI crash testing and formally define GUI crash tests. In Section 3, we present three empirical studies demonstrating the usefulness of crash tests. In Section 4, we discuss related work, and finally in Section 5 conclude with a discussion of ongoing and future work.

2 GUI Crash Tests

Users interact with a GUI by performing *events* on some widgets, such as clicking on a button, opening a menu, and dragging an icon. During conventional GUI testing, test cases, consisting of sequences of events are executed on the GUI.[4] Our goal now is to create test cases on-the-fly that can quickly test major parts of the GUI fully automatically. More specifically, we want to produce test cases that satisfy the following requirements.

- The crash test cases should be generated quickly on-the-fly and executed. We don't intend to save the test cases as a suite; rather, we want a throwaway set of test cases that require no maintenance.
- The test cases should broadly cover the GUI's entire functionality.
- It is expected that new changes will be made to the GUI before the crash testing process is complete. Hence, we will terminate and restart the crash testing process each time a new change is checked-in. The crash test cases should detect major problems in a short time interval.

We develop the crash test cases automatically using model-based techniques. The key reason for our success is a

[4] We have shown in earlier work that simply executing each event in isolation is not enough for effective GUI testing [25].

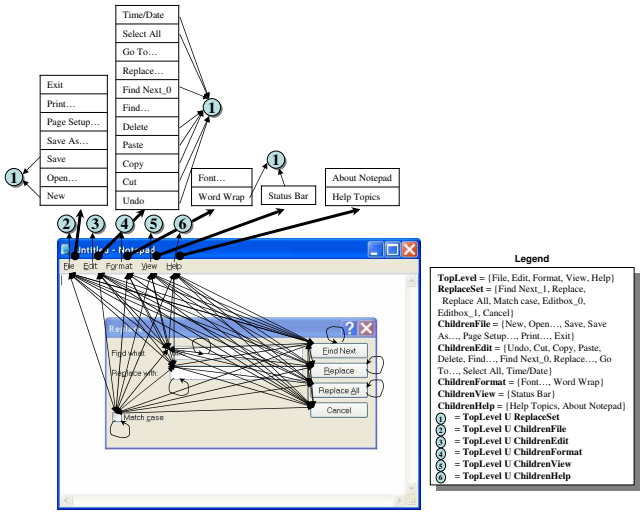

Figure 1. Example of an Event-Flow Graph.

new GUI representation that models specialized user interactions with the GUI. We call this representation the *event-interaction graph*, which is based on a structure called the event-flow graph (EFG) [21].

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

EFGs may be used to generate GUI test cases. A straightforward way to generate test cases is to start from a known initial state of the GUI (*e.g.*, the state in which the software starts) and use a graph traversal algorithm, enumerating the nodes during the traversal, on the EFG. If the event requires text input, *e.g.*, for a text-box, then its value is read from a database, initialized by the software tester. A sequence of events $e_1; e_2; \ldots; e_n$ is generated as output that serves as a GUI test case. For example, the EFG of Figure 1 may be used to generate a number of test cases, *e.g.*, $< File; FindNext; File >$ and $< New; Format; Font >$.

This straightforward approach works well in certain situations; we have used it to generate test suites for large empirical studies [27, 21]. However, the number of event sequences grows very rapidly with length. It becomes infeasible to generate and execute all possible event sequences beyond $length > 2$. In previous work, we have always executed a reasonable subset. We have learned that many of the longer sequences are useful for fault-detection [26]. The challenge is to automatically generate these effective sequences. In this research, we do this by further reducing the event interaction space using lessons that we have learned from our empirical studies. We now model a reduced event-interaction space by developing event-interaction graphs.

We have observed that in a typical GUI, 20-25% of the GUI events manipulate the structure of the GUI; examples include events that open/close windows/menus. For example, in Microsoft Word, of the total 4210 events, 80 events open menus, 346 events open windows, and 196 events close windows; the remaining 3588 events interact with the underlying code. The code for events that open menus and windows is straightforward, usually generated automatically by visual GUI-building tools. Our experience with GUI testing has shown that this code is very unlikely to interact with code of other events; hence very few errors are revealed by executing interactions between these events. Our goal is to test interactions between the remaining (non-structural) events. We automatically identify these events, model the interactions between them, and reduce the space of interactions that need to be tested.

We identify windowing events that open windows (*e.g.*, the event `Open` used to open the window `File Open`) and *termination events* that close windows; common examples include `Ok` and `Cancel`. The GUI contains other types of events that do not open or close windows but make other GUI events available. For example, *menu-open events* are used to open menus. The most common example of menu-open events are generated by widgets that open pull-down menus. For example, `File` and `Edit` found in many applications' pull-down menus are menu-open events.

The above events are structural, *i.e.*, they manipulate the structure (open/close menus/windows) of the GUI. The remaining *non-structural events* do not cause structural changes to the GUI; rather they are used to perform some action; common examples include the `Copy` event used for copying objects to the clipboard.

Note that events that interact with the underlying software include non-structural and termination events. We call these events system-interaction events. Intuitively, our GUI crash test cases are composed only of these events (and any other events necessary to "reach" the system-interaction

events). Also, we focus on all two-way interactions between these events.

We now define some terms that we will use to develop event-interaction graphs. An event-flow path represents a sequence of events that can be executed on the GUI. Formally, an event-flow-path is defined as follows:

Definition: There is an *event-flow-path* from node n_x to node n_y iff there exists a (possibly empty) sequence of nodes $n_j; n_{j+1}; n_{j+2}; \ldots; n_{j+k}$ all in the event-flow graph E such that $\{(n_x, n_j), (n_{j+k}, n_y)\} \subseteq edges(E)$ and $\{(n_{j+i}, n_{j+i+1}) for\ 0 \leq i \leq (k-1)\} \subseteq edges(E)$. □

The function $edges$ takes an EFG as an input and returns a set of ordered-pairs, each representing an edge in the EFG. We use the notation $< n_1; n_2; \ldots; n_k >$ for an event-flow path. Several examples of event-flow paths from the EFG of Figure 1 are: $< File; Edit; Undo >$, $< File; MatchCase; Cancel >$, $< MatchCase; Editbox\_1; Replace >$, and $< MatchCase; FindNext; Replace >$. We are interested in those event-flow-paths that start and end with system-interaction events, without any intermediate system-interaction events.

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

Of the examples of event-flow paths presented above, $< File; Edit; Undo >$ is interaction-free (since $Edit$ is not a system-interaction event) whereas $< MatchCase; Editbox\_1; Replace >$ is not (since $Editbox\_1$ is a system-interaction event).

We now define the *interacts-with* relationship between system-interaction events. Intuitively, two system-interaction events may interact if a GUI user may execute them in an event sequence without executing any other intermediate system-interaction event.

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

For the EFG of Figure 1, the above relation holds for the following pairs of system-interaction events: $\{(New, Date/Time), (FindNext\_1, WordWrap), (Editbox\_0, Editbox\_1)$, and $(Delete, Cancel)\}$. The interaction-free event-flow-paths for these pairs are $< New; Edit; Date/Time >$, $< FindNext\_1; Format; WordWrap >$, $< Editbox\_0; Edibox\_1 >$, and $< Delete; Cancel >$ respectively. Note that an event may interact-with itself. For example, the event `MatchCase` interacts with itself. Also note that "e_x interacts-with e_y" does not necessarily imply that "e_y interacts-with e_x." For example, in our EFG, even though $Replace$ interacts-with $Cancel$, the event $Cancel$ does not interact-with $Replace$.

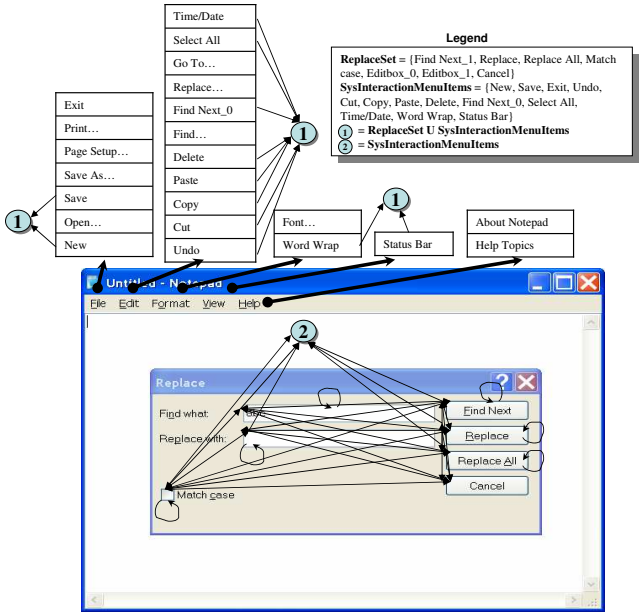

Figure 2. EIG for the EFG of Figure 1.

We use the interacts-with relationship to create the event-interaction graph (EIG). This graph contains nodes, one for each system-interaction event in the GUI. An edge from node n_x (that represents e_x) to node n_y (that represents e_y) means that e_x interacts-with e_y.

The event-interaction graph for the EFG of Figure 1 is shown in Figure 2. Note that the space of event-sequences has reduced considerably since only the system-interaction event interactions are marked in this graph.

We may traverse the event-interaction graph in a number of ways to generate sequences of system-interaction events. For example, we may generate all length 1 event sequences by simply enumerating all the nodes in the graph. For the event-interaction graph of Figure 2, we obtain the set of length 1 sequences $\{New, Save, Undo, Cut, Copy, Paste, Delete, FindNext\_0, SelectAll, Time/Date, WordWrap, StatusBar, MatchCase, Editbox\_0, Editbox\_1, FindNext\_1, Replace, ReplaceAll, Cancel\}$. We may generate all length 2 event sequences by enumerating each node with its adjacent node, *i.e.*, each edge in the EIG. For the event-interaction graph of Figure 2, we obtain sequences such as $< New; Undo >$, $< FindNext\_1; Replace >$, and $< ReplaceAll; Exit >$.

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

cution.

The test cases defined in the previous section are complete, in that they can be executed automatically on the GUI. Crashes during test execution may be used to identify serious problems in the software.

Once we have obtained the event-interaction graph for a GUI, we can annotate it in several ways. In this paper, we associate a boolean flag with each edge in the graph. During crash testing, once we have generated a test case that "covers" an edge, we set the associated boolean flag. This prevents us from generating this test again, until all the edges have been covered.

If the crash testing process is interrupted, *e.g.*, when a new version of the software has been checked-in, the flags for each edge are retained across event-interaction graph versions. This allows us to implement algorithms that rotate test cases across software versions.

3 Empirical Studies

We have implemented the algorithms described in the previous section as modules of a new system called GUICrasher. GUICrasher is able to automatically analyze the GUI, create event-interaction graphs, generate crash test cases, and execute them. The *GUI ripper* is the automated module that creates the event-interaction graphs [23]. "GUI Ripping" is a dynamic process in which the software's GUI is automatically "traversed" by opening all its windows and extracting all their widgets (GUI objects), properties, and values. The *test-case generator* uses the event-interaction graphs to create the crash tests. The *test executor* is capable of executing the test cases automatically on the GUI. It performs all the events in each test case. Events are triggered on the GUI using the native OS API. For example, the windows API *SendMessage* is used for windows applications and Java API *doClick* for Java applications. Note that we have provided only the details needed to understand the empirical studies and interpret the results. Additional details and algorithms are available in [21, 25].

We now present three empirical studies to demonstrate the usefulness of our crash tests. In particular, we will answer the following questions:

1. How many times does a typical software crash on our test cases?
2. How long does it take to run our crash tests?
3. Since the crash testing process is expected to be terminated as soon as the GUI is modified again, which could give a very small window of time to run the test cases, how many test cases must be run to completion for effective testing?
4. When rotating test cases during frequent GUI modification, how effective is the annotated event-interaction graph approach?

3.1 Study 1: Effectiveness of Crash Tests

The goal of our first empirical study is to determine the feasibility of the crash testing process and the effectiveness of the tests. In particular, we use the following process:

1. Choose software subjects with GUI front-ends.
2. Generate event-interaction graphs.
3. Generate crash test cases on-the-fly, executing each automatically on the subject applications.

We measure the time taken to execute the test cases and the number of software crashes reported.

Step 1: Software Subjects: The software subjects for our studies are part of an open-source office suite developed at the Department of Computer Science of the University of Maryland by undergraduate students of the senior Software Engineering course. It is called TerpOffice[5] and consists of six applications out of which we use four – TerpSpreadSheet (a spreadsheet application), TerpPaint (an image editing/manipulation program), TerpPresent (a presentation tool), and TerpCalc (a scientific calculator with graphing capability). They have been implemented using Java. Table 1 summarizes the characteristics of these applications. Note that these applications are fairly large with complex GUIs. With the exception of TerpCalc, all the applications are roughly the size of MS WordPad. The number of widgets listed in the table are the ones on which system-interaction events can be executed. We did not include the `Help` menu, since the help application is launched in a separate web browser.

Subject Application	Windows	Widgets	LOC	Classes	Methods	Branches
TerpSpreadSheet	9	145	12791	125	579	1521
TerpPaint	10	200	18376	219	644	1277
TerpCalc	1	82	9916	141	446	1306
TerpPresent	12	294	44591	230	1644	3099
TOTAL	32	721	85674	715	3313	7203

Table 1. TerpOffice Applications

Step 2: Generate System-Interaction Graphs: For each application, we used GUICrasher to generate event-interaction graphs. The sizes of the event-interaction graphs are shown in Table 2. As noted earlier, our crash tests will consist of all length 1 (number of events in the EIG) and length 2 (number of edges in the EIG) system-interaction event sequences. Hence, in this study, we expect to generate and execute more than 39K crash tests.

Subject Application	Nodes	Edges
TerpSpreadSheet	145	3246
TerpPaint	200	8699
TerpCalc	82	6561
TerpPresent	294	19918
TOTAL	721	38424

Table 2. Sizes of Event-Interactions Graphs

Step 3: Test-Case Generation and Execution: GUICrasher generated all the crash test cases and replayed them on the

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

subject applications one-by-one. The execution consisted of performing each event, such as a clicking-on-buttons, opening-menus, selecting-items, checking-boxes, etc. If a text-box needed input, then the values were read from a database. We initialized the database to contain several types of inputs: negative and positive integers, text strings, special characters, very long strings, and floating-point numbers.

The time needed (in minutes) to run all the test cases is shown in Figure 3. We see that it took approximately 3-6 hours to execute all the crash test cases for TerpCalc, TerpSpreadSheet and TerpPaint, and 12 hours for TerpPresent. We will see later, in Studies 2 and 3, that not all the test cases need to be executed to reveal problems in the software.

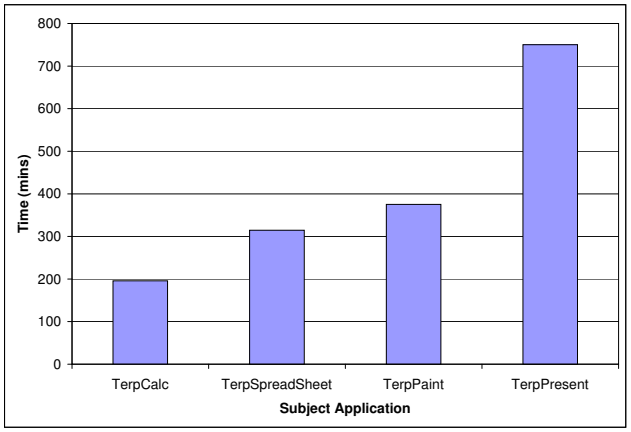

Figure 3. Total Execution Time

Results: Crashes Reported: We now see the effectiveness of the crash tests. Figure 4 shows the total number of test cases that led to a software crash. The large number of crashes reported was very encouraging, especially since this version of TerpOffice was considered to be stable and has a test suite of 5000+ GUI test cases; it also has at least one JUnit test case per Java method.

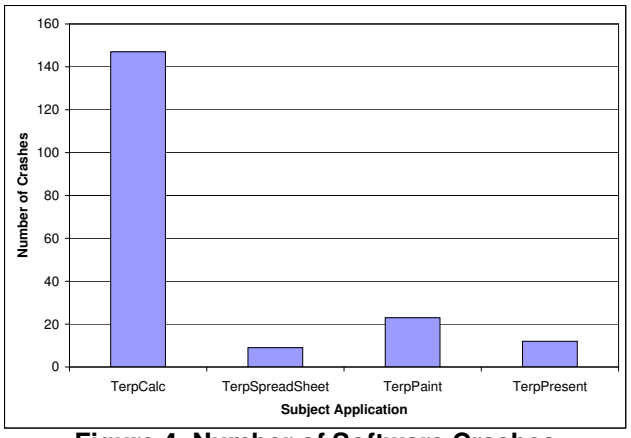

Figure 4. Number of Software Crashes

We manually examined all the system exception messages and computed the number of bugs (we informally use the term "bug" to mean a "fault in the code") in the code that had led to the crashes. Figure 5 summarizes our results. We see that although TerpCalc had crashed on a large number (140+) of test cases, the crashes were due to only 3 bugs in the code. Also, TerpPaint had crashed on only 23 test cases but the number of underlying bugs was 13, a surprisingly large ratio. The ratios between crashes and bugs are in fact due to the location of the bugs; if frequently executed code contains the crash-causing bug, then a large number of test cases will result in a crash.

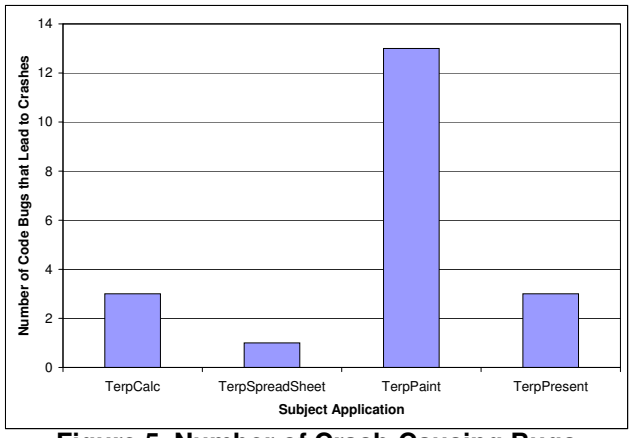

Figure 5. Number of Crash-Causing Bugs

We now describe some of the bugs that manifested themselves as software crashes. We identified three classes: (1) *Invalid text input.* We found that many crashes were the result of the software not checking the validity and size of text input. For example, some text boxes in TerpPaint expect an integer input; providing a string resulted in a crash. In some instances, a "very long" text input also resulted in a crash. (2) *Widget enabled when it should be disabled.* One challenge in GUI design is to identify allowable sequences of interactions with widgets and to disallow certain sequences. Designers often disable certain widgets in certain contexts. In our TerpOffice applications, we found several instances of widgets enabled when they should really have been disabled. When our crash tests executed the incorrectly enabled widget in an event sequence, the software crashed. (3) *Object declared but not initialized.* Some of our crashes were Java NullPointerExceptions. It turned out that as the software was evolving, one developer, not seeing the use of an object, commented out a part of the code, which was responsible for object initialization. Another developer continued to use the object in another part of the code. The software crashed when the uninitialized object was accessed.

3.2 Study 2: Number of Crash Tests

From our previous study, we saw that running all crash tests can take up to 12 hours. Since we may not have such

a long time between GUI code changes, we conducted another study to determine the impact of number of test cases on the number of crash-causing bugs detected. Since we already had the results of the crash test cases from the previous study, we did not have to regenerate and re-execute new test cases for this study. We could simply simulate the effect of different number of test cases by treating our existing smoke test suite as a *test pool* and selecting different number of test cases from them. More specifically, for each subject application, we used the test pool to create 1200 test suites: 200 of each size 100, 500, 1000, 2000, 3000, and 4000. Each suite was obtained independently using random selection without replacement.

Since we have 200 test suites of each size, we show our results in the form of box-plots. The box-plots provide a concise display of each distribution. The black square inside each box marks the median value. The edges of the box mark the first and third quartiles. The whiskers extend from the quartiles and cover the entire distribution. The median in the box-plots of Figure 6 shows that the number of bugs revealed increases with test suite size; however, as the overlaps between box-plots show, the number of bugs does not grow significantly with test suite size. We also conduced a one-way ANOVA with factor "test suite size" and response "number of bugs revealed." We found no statistically significant impact of suite size on the number of bugs revealed. Hence, even a small number (a few hundreds) of crash tests are sufficient to find serious software problems.

3.3 Study 3: Rotating Test Cases

Next we wanted to see the effectiveness of our rotating algorithm based on the annotated event-interaction graphs. Recall that the algorithm ensures that all crash tests are executed across multiple code changes. In this study, we perform the following steps:

1. Start with the original (faulty) subject applications. We already know the number of crash-causing bugs in each application.
2. Set a time interval $\mathcal{N}$ between software changes.
3. Generate and execute as many crash test cases as possible in this interval. Update the boolean flags on the event-interaction graph edges.
4. Examine the crashes reported and eliminate the revealed bugs.
5. Repeat Steps 3 and 4 until there are no more bugs.

We used a random selection without replacement as a control strategy for this study, *i.e.*, instead of using the boolean-edge information, we randomly selected a crash test case, making sure that each test case was selected only once in one interval. Also, we used four values of $\mathcal{N}$, *i.e.*, 15, 30, 60, and 90 minutes. We repeated this study 200 times and report results of medians.

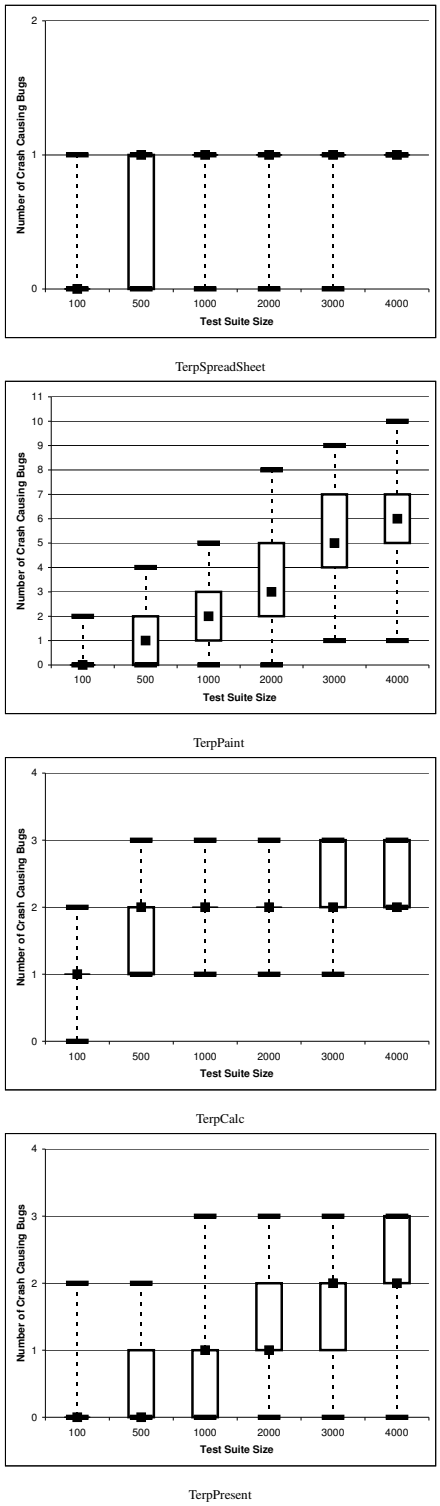

Figure 6. Number of Bugs vs. Number of Test Cases

We summarize the results of this study in Figure 7. Each graph has two lines,[6] one for our control "Random" and the other for our "Memory"-based on the boolean flags in the event-interaction graphs. The x-axis shows the intervals between code changes, the first one being the start interval 0. The y-axis shows the number of bugs remaining in the subject applications. Note that we do not show the results for TerpSpreadSheet, since it has only one bug, making its results uninteresting.

As the graphs show, the memory-based rotation technique does better than the random technique, *i.e.*, we remove all bugs much sooner. In the case of TerpPaint, the random technique does not even eliminate all bugs.

4 Related Work

To the best of our knowledge, there is no reported work on rapid QA of evolving GUI software. There are several areas that share characteristics relevant to our work: feedback-based QA mechanisms for conventional software, GUI testing tools, robustness testing and fault injection. We discuss some of these topics here. Note that we have discussed GUI testing tools in our other reported work on smoke testing [21]; we will not reproduce it here for lack of space.

Feedback-based QA mechanisms are increasingly being used to control the quality of evolving software. We classify these mechanisms according to their frequency and speed of execution, thoroughness, and the level of detail of feedback that they provide to a developer: (1) *Immediate*, (2) *Intermediate*, and (3) *Thorough*. The *immediate* mechanisms are usually integrated with version control systems to provide immediate feedback at change-commit time. The developer manually provides a commit validation suite that is executed each time a new software version is checked-in. The commit-validation test suite typically consists a few test cases. Systems such as Aegis implement a two-stage commit that first does a "partial commit" of the changes, executes test cases, and does a "final commit" only after all the test cases have passed.

The *intermediate* mechanisms include daily building and smoke testing that executes more thorough test cases on a more regular basis. Developers don't get instant feedback; rather they are e-mailed the results of the nightly builds and smoke tests. Usually a software crash or a mismatch will be considered as a bug. During nightly builds, a development version of the software is checked out from the source code repository tree, compiled, linked and "smoke tested", with the purpose to (re)validate the basic functionality of the system [19]. A number of large-scale commercial and open-source projects apply daily building and smoke testing, including Microsoft Windows NT operating system [20], and many GNU projects, such as *Ghostscript*, *Mozilla*, *open-webmail*, *WINE*, etc. There are several tools that may be used to setup and perform smoke testing of software applications. Most of these tools provide more or less identical functionality for conventional software. Popular examples include *CruiseControl* [2], *IncrediBuild* [3], *Daily Build* [5] and *Visual Build* [4]. Our own system called DART [27, 21] addresses the needs of smoke testing of GUI software.

The *thorough* mechanisms such as Skoll [22] continuously run hundreds and thousands of test cases for weeks at a time on several builds of the evolving software. Several Skoll *servers* manage the distribution of test cases on dozens of Skoll *clients* that run the test cases. Results and feedback is collected by the servers and analyzed. Decisions about subsequent test executions are then made.

Several researchers have conducted research to increase the **robustness** of software [15, 11, 12], although not in the context of continuous testing. The overall goal of robustness testing is to ensure that the program handles all (valid and invalid) inputs. Most of the research in this area focuses on the generation of robustness-checking test cases. **Fault injection** techniques may also be used to test the robustness of a program [7]. The injectors introduce the faults into the application by modifying the code at compile time [14, 30] and runtime [18, 13] or by introducing faults in its execution environment [10, 17]. The goal is to force various types of failures such as memory corruption [8, 6].

5 Conclusions

One of the major challenges of developing rapidly evolving software is maintaining its quality. While there are several feedback-driven mechanisms to control the quality of conventional evolving software, there are no such mechanisms for GUI-based applications. This paper makes several contributions to the field of rapidly evolving GUI software. First, it identifies the requirements for rapid GUI testing. Second, it presents a new type of GUI testing, called "crash testing" to help rapidly test the GUI as it evolves. Third, it presents two levels of crash testing: (1) immediate feedback-based in which a developer indicates that a GUI bug was fixed in response to a previously reported crash; only the select crash test cases are rerun and the developer is notified of the results in a matter of seconds, and (2) between code changes in which new crash test cases are generated on-the-fly and executed on the GUI. Fourth, since the code may be changed by another developer before all the crash tests have been executed, hence requiring restarting of the process, it presents a simple rotation-based scheme to ensure that all crash tests are executed over a series of code changes. Finally, it presents empirical studies that demonstrate the effectiveness of the crash tests.

Our research on crash testing continues; this paper is just the first report in a line of research that looks promising with many exciting opportunities for extensions. First, we are exploring the use of fault-injection techniques to improve

[6] If only one line is visible, they are overlapping

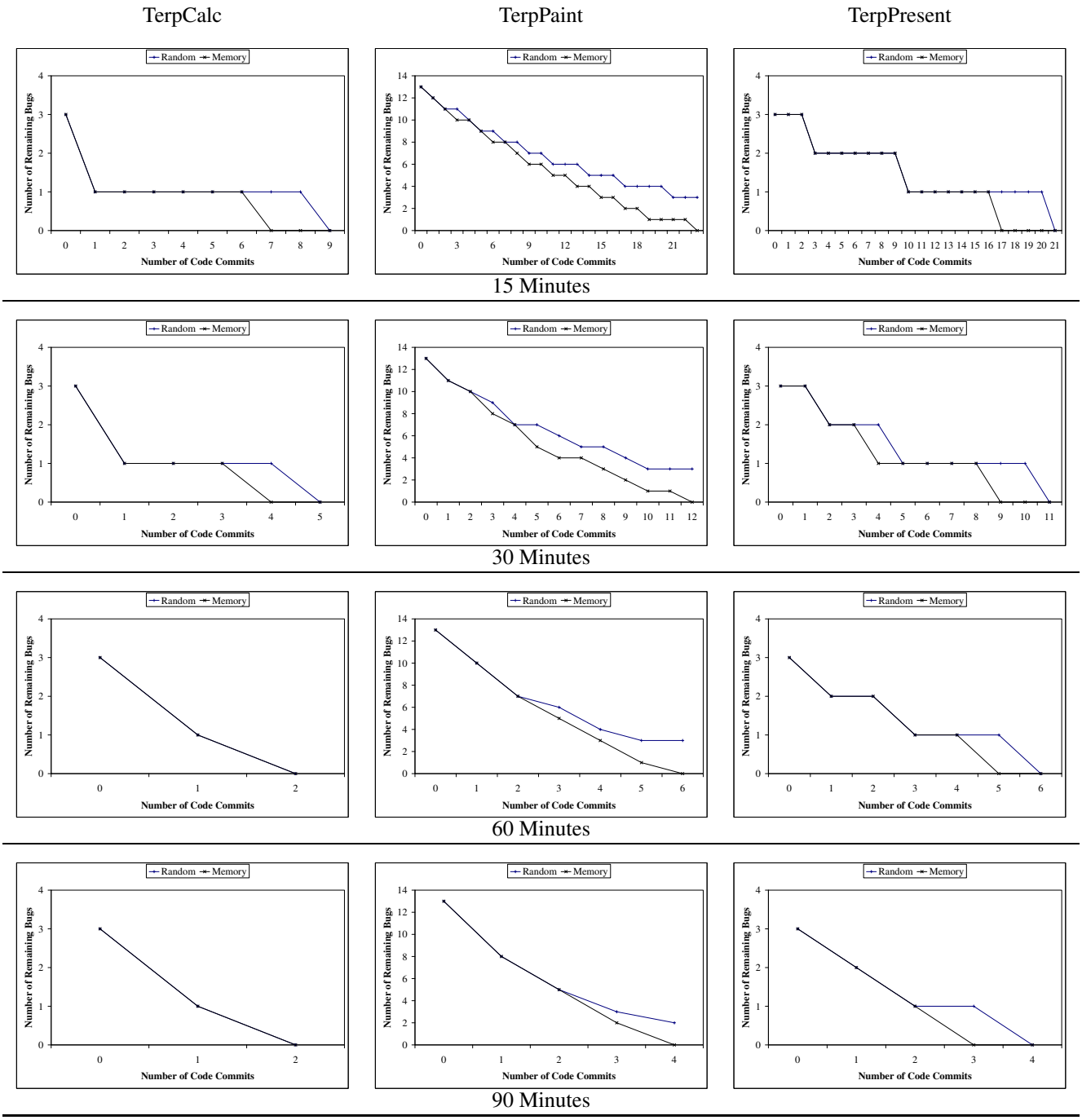

Figure 7. Effectiveness of the Rotating Algorithm. Note that the "Memory" line touches the x-axis faster than the "Random" line, indicating that crash-causing bugs are eliminated faster when using the annotated event-interaction graph approach.

the crash tests. Second, we are studying the effectiveness of longer crash tests and their impact on the number of bugs revealed. Third, we are extending this work to other reactive software systems, namely web applications. Finally, we are developing a web service which will enable users to submit their GUI-based applications for testing; GUICrasher will "crash test" the application and report results of crashes found. We will integrate this web service into the development of TerpOffice in future Software Engineering courses.

References

[1] ACE+TAO software release. deuce.doc.wustl.edu/Download.html.

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

[3] FAST C++ Compilation - IncrediBuild by Xoreax Software, 2003. http://www.xoreax.com/main.htm.

[4] Kinook Software - Automate Software Builds with Visual Build Pro, 2003. http://www.visualbuild.com/.

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

[6] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. Goofi: Generic object-oriented fault injection tool. In *Proceedings of the International Conference on Dependable Systems and Networks, 2001*, pages 83–88, 2001.

[7] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell. Fault injection for dependability validation: A methodology and some applications. *IEEE Trans. Softw. Eng.*, 16(2):166–182, 1990.

[8] J. H. Barton, E. W. Czeck, Z. Z. Segall, and D. P. Siewiorek. Fault injection experiments using fiat. *IEEE Trans. Comput.*, 39(4):575–582, 1990.

[9] H. F. Brophy. Improving programming performance. *Australian Computer Journal*, 2(2):66–70, 1970.

[10] J. Carreira, H. Madeira, and J. G. Silva. Xception: A technique for the experimental evaluation of dependability in modern computers. *IEEE Transactions on Software Engineering*, 24(2):125–136, Feb. 1998.

[11] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for java. *Software Pracetice and Experience*, 34:1025–1050, 2004.

[12] M. Dix and H. D. Hofmann. Automated software robustness testing - static and adaptive test case design methods. In *Proceedings of the 28th Euromicro Conference*, pages 62–66, 2002.

[13] J.-C. Fabre, M. Rodríguez, J. Arlat, and J.-M. Sizun. Building dependable cots microkernel-based systems using mafalda. In *Proceedings of Pacific Rim International Symosium on Dependable Computing, 2000*, pages 85–94, 2000.

[14] C. Fetzer, P. Felber, and K. Högstedt. Automatic detection and masking of nonatomic exception handling. *IEEE Trans. Software Eng.*, 30(8):547–560, 2004.

[15] C. Fu, B. G. Ryder, A. Milanova, and D. Wonnacott. Testing of java web services for robustness. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 23–34. ACM Press, 2004.

[16] J. D. Herbsleb and D. Moitra. Guest Editors' introduction: Global software development. *IEEE Software*, 18(2):16–20, Mar./Apr. 2001.

[17] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. Ferrari: A tool for the validation of system dependability properties. In *Proceedings of 22nd International Symposium on Fault-Tolerant Computing*, pages 336–344, 1992.

[18] N. P. Kropp, P. J. K. Jr., and D. P. Siewiorek. Automated robustness testing of off-the-shelf software components. In *Proceedings of 28th International Symposium on Fault-Tolerant Computing*, pages 230–239, 1998.

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

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

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

[22] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. C. Schmidt, and B. Natarajan. Skoll: Distributed Continuous Quality Assurance. In *Proceedings of the 26th IEEE/ACM International Conference on Software Engineering*, Edinburgh, Scotland, May 2004. IEEE/ACM.

[23] A. M. Memon, I. Banerjee, and A. Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *Proceedings of The 10th Working Conference on Reverse Engineering*, pages 260–269, Nov. 2003.

[24] 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.

[25] 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.

[26] A. M. Memon and M. L. Soffa. Regression testing of GUIs. In *Proceedings of the 9th European Software Engineering Conference (ESEC) and 11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-11)*, pages 118–127, Sept. 2003.

[27] A. M. Memon and Q. Xie. Empirical evaluation of the fault-detection effectiveness of smoke regression test cases for gui-based software. In *Proceedings of The International Conference on Software Maintenance 2004 (ICSM'04)*, pages 8–17, Chicago, Illinois, USA, Sept. 2004.

[28] R. Prikladnicki and J. L. Nicolas. Requirements engineering in global software development: Preliminary findings from a case study in a SW-CMM context. 2003.

[29] R. D. Riecken, J. Koenemann-Belliveau, and S. P. Robertson. What do expert programmers communicate by means of descriptive commenting? In *Empirical Studies of Programmers: Fourth Workshop*, Papers, pages 177–195, 1991.

[30] H. A. Rosenberg, K. G. Shin, and S. Han. DOCTOR: An integrated software fault injection environment for distributed real-time systems. In *Proceedings of the International Computer Performance and Dependability Symposium 1995*, pages 204–213, april 1995.

[31] C. B. Seaman and V. R. Basili. Communication and organization: An empirical study of discussion in inspection meetings. *IEEE Transactions on Software Engineering*, 24(7):559–572, July 1998.

[32] C. R. B. D. Souza. Global software development: Challenges and perspectives, May 13 2001.