# Model-Based Testing of Community-Driven Open-Source GUI Applications

Qing Xie and Atif M. Memon
Department of Computer Science
University of Maryland, College Park, MD 20742
{qing, atif}@cs.umd.edu

**Abstract**

Although the world-wide-web (WWW) has significantly enhanced open-source software (OSS) development, it has also created new challenges for quality assurance (QA), especially for OSS with a graphical-user interface (GUI) front-end. Distributed communities of developers, connected by the WWW, work concurrently on loosely-coupled parts of the OSS and the corresponding GUI code. Due to the unprecedented code churn rates enabled by the WWW, developers may not have time to determine whether their recent modifications have caused integration problems with the overall OSS; these problems can often be detected via GUI integration testing. However, the resource-intensive nature of GUI testing prevents the application of existing automated QA techniques used during conventional OSS evolution. In this paper we develop new process support for three *nested* techniques that leverage developer communities interconnected by the WWW to automate model-based testing of evolving GUI-based OSS. The "innermost" technique (*crash testing*) operates on each code check-in of the GUI software and performs a quick and fully automatic integration test. The second technique (*smoke testing*) operates on each day's GUI build and performs functional "reference testing" of the newly integrated version of the GUI. The third (outermost) technique (*comprehensive GUI testing*) conducts detailed integration testing of a major GUI release. An empirical study involving four popular OSS shows that (1) the overall approach is useful to detect severe faults in GUI-based OSS and (2) the nesting paradigm helps to target feedback and makes effective use of the WWW by implicitly distributing QA.

## 1 Introduction

Open-source software (OSS) development and evolution has greatly benefited from the pervasive nature of the world-wide web (WWW). Today, OSS is typically developed and maintained by communities of programmers distributed world-wide. Consequently, the WWW allows OSS developers to use the world's different time-zones to their advantage, thereby enabling continuous around-the-clock software evolution that "follows the sun" [5]. This practice of around-the-clock evolution has led to unprecedented OSS code churn rates. For example, the OSS ACE+TAO [1] developers average 200+ CVS commits per week [8].

While successful at increasing code churn rates, web-based community driven OSS evolution suffers from several problems. First, there is little direct inter-developer communication [16]. Almost all communication is done via web-based tools such as CVS commit log messages, bug reports, change-requests, and comments [3, 15]. Second, sub-groups within developer communities often work on loosely coupled parts of the application code [16]. Each developer (sub-group) typically modifies a local "copy" of the code and frequently checks-in changes (and down-loads other developers' changes). Consequently, after making a change, a developer may not immediately realize that the local change has inadvertently broken other parts of the overall software code [7]. In such situations, the developer needs quick feedback of newly introduced faults, enabling quick fixes. If left undetected, the cascading effect of these faults leads to wasted debugging cycles during development and expensive quality assurance later. Moreover, intermediate fielded releases of the OSS have questionable quality.

Graphical-user interfaces (GUIs) are one of the most important parts of software [17]. Although GUIs are popular and useful, developer communities of GUI-based OSS, interconnected by the WWW, face severe QA challenges. These challenges stem from the fact that loosely coupled parts of the OSS are being modified rapidly by several developers simultaneously; whenever interfaces to these parts change, the developers also modify the GUI, which is common to the entire OSS. In many OSS, the GUI is the only place where several loosely coupled parts of the code interact. Due to the unprecedented code-churn rates enabled by the WWW, developers may have little or no time to determine whether their local modifications have inadvertently broken other parts of the OSS code. For example, while several developers may be working on different parts of a large desktop application (*e.g.*, the Print module, text-search/replace, Preferences setting), the GUI brings all these parts together; interactions between these parts usually lead to problems. Such problems can often be detected

by performing GUI integration testing. The feedback-based mechanisms mentioned earlier are not easily adaptable for automated GUI testing since GUIs have characteristics different from those of traditional software; techniques typically applied to software testing are not adequate.

This paper presents a new technique with supporting tools and processes that leverage the WWW and developer community for continuous integration testing of GUI-based applications. The key idea of this technique is to create concentric testing loops, each with specific GUI testing goals, resource usage, and target developer sub-groups for error reports. Instances of three such loops are presented. The tightest loop called the *crash testing* loop operates on each code check-in (*e.g.,* using CVS) of the GUI software [19]. It is executed very frequently and hence is designed to be very inexpensive. The goal is to perform a quick-and-dirty, fully automatic integration test of the GUI software. Software *crashes* are reported back to the developer who checked-in the code. The second loop is called the *smoke testing* loop which operates on each day's GUI build [7, 12]. It is executed nightly/daily and hence is designed to complete within 8-10 hours. The goal of this loop is to do functional "reference testing" of the newly integrated version of the GUI. Differences between the outputs of the previous (yesterday's) build and the new build are reported to developers who made changes since the last smoke test run. These two loops may be implemented as web services. The third, and outermost loop is called the "comprehensive GUI testing" loop. It is executed after a major version of the GUI is available. The goal of this loop is to conduct comprehensive GUI integration testing, and hence is the most expensive. Problems in the GUI software are reported to all the developers who contributed to the release. A novel feature of the continuous testing technique is a GUI model that is obtained automatically using reverse engineering techniques [10]. This model is used to generate test cases, create descriptions of expected execution behavior, and evaluate the adequacy of the generated test cases. Automated test executors "play" these test cases on the GUI and report errors.

All the GUI testing loops have been implemented. In earlier work, we showed that comprehensive model-based testing is useful in that it helps to detect OSS integration problems via the GUI [11, 19], and that smoke testing is practical and useful [7, 12]. In our experience, putting these techniques together helps to conserve resources, *i.e.*, faster "inner" techniques help to detect GUI faults quickly. These faults, if left undetected, would have delayed the outer loops. We now evaluate crash testing in an experiment in Section 5 on four popular OSS developed by communities of developers on the WWW. The results of the experiment show that (1) the concentric loops provide an effective mechanism for resource utilization; errors that would otherwise delay the outer loops are caught earlier by the inner loops and (2) popularly used GUI-based OSS have serious flaws that persist across multiple versions; these flaws could have been detected by our approach fully automatically. We have reported these flaws to the developers of these applications; they have fixed them in new releases. Note that this work builds upon six years of work on model-based GUI testing. To conserve space, we will not discuss the details of the models and associated algorithms; we will however give a brief overview. The interested reader is referred to related literature [7, 9–12, 19].

The specific contributions of this paper include:

- Recognition that the nature of the WWW enables the separation of GUI testing steps by level of automation, feedback, and resource utilization.
- Three types of processes that leverage WWW developer communities for distributed QA of GUI-based OSS.
- Demonstration that resources may be better utilized by defining a concentric loop-based GUI testing approach.
- Demonstration that popular GUI-based OSS developed on the WWW have flaws that can be detected by our fully-automated approach.

**Structure of the paper:** The next section discusses related work. Section 3 provides an overview of three process types that support the continuous GUI QA. The model that is central to these process types is described in Section 4. Section 5 presents an experiment that demonstrate the usefulness of the continuous integration testing technique on several GUI-based OSS.

## 2 Background & Related Work

There are several feedback-based mechanisms to help manage the quality of OSS developed by communities of developers on the WWW. These mechanisms improve the quality of OSS via continuous, rapid quality assurance (QA) during evolution. They differ in the level of detail of feedback that they provide to targeted developers, 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 (web-enabled) validation steps when checking into the NetBeans CVS repository.[1] In fact, some web-based 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)

[1] http://www.netbeans.org/community/guidelines/commit.html
[2] http://aegis.sourceforge.net/

basis at central server sites. Developers do not get instant feedback; rather they are e-mailed the results of the nightly builds and smoke tests. Another, still higher level of continuous QA support is provided by mechanisms such as Skoll [8] that continuously run test cases, for days and even weeks on several builds (stable and beta) of the evolving OSS using user-contributed resources over the WWW. For example, the ACE+TAO OSS is tested continuously by Skoll; results are summarized in a web-based virtual scoreboard.[3] All these mechanisms are useful, in that they leverage the WWW to detect defects early during OSS evolution. Moreover, since the feedback is directed towards specific developers (*e.g.*, those who made the latest modifications), QA is implicitly and efficiently distributed.

Testing the correctness of a GUI is difficult for a number of reasons. First of all, the space of possible interactions with a GUI is enormous, in that each sequence of GUI events can result in a different state, and each GUI event may need to be evaluated in all of these states [11]. The large number of possible states results in a large number of input permutations [17] requiring extensive testing. A related problem is to determine the coverage of a set of test cases [9]. For conventional software, coverage is measured using the amount and type of underlying code exercised. These measures do not work well for GUI testing, because what matters is not only how much of the code is tested, but in how many different possible states of the software each piece of code is tested. An important aspect of GUI testing is verification of its state at each step of test case execution [9]. An incorrect GUI state can lead to an unexpected window/screen, making further execution of the test case useless since events in the test case may not match the corresponding GUI elements on the screen. Thus, the execution of the test case must be terminated as soon as an error is detected. Also, if verification checks are not inserted at each step, it may become difficult to identify the actual cause of the error. Finally, regression testing presents special challenges for GUIs, because the input-output mapping does not remain constant across successive versions of the software [9, 13].

The most common way to test a GUI-based OSS is to wait until the software's GUI has "stabilized," typically before a major release is planned. Some developers (or testers) in the community then use capture/replay tools [6] such as WinRunner[4] [12] to test the new *major GUI version release*. A tester uses these tools in two phases: a capture and then a replay phase. During the capture phase, a tester manually interacts with the GUI being tested, performing events. The tool records the interactions; the tester also visually "asserts" that a part of the GUI's response/state be stored with the test case as "expected output" [18]. The recorded test cases are replayed automatically on (a modified version of) the GUI using the replay part of the tool. The "assertions" are used to check whether the GUI executed correctly. Another way to test a GUI is by *programming* the test cases (and expected output) using tools [4, 17] such as extensions of *JUnit* including *JFCUnit*, *Abbot*, *Pounder*, and *Jemmy Module*[5]. The above techniques require a significant amount of manual effort, typically yielding a small number of test cases. The result is an inadequately tested GUI [9]. Moreover, during iterative development, developers waste time fixing bugs that they encounter in later development cycles; these bugs could have been detected earlier if the GUI had been tested iteratively. The nature of community-driven GUI development requires new GUI testing techniques to quickly test each increment of the GUI during development.

The most comprehensive and complete solution for GUI testing is provided by GUITAR [7, 12] that automates testing of software applications that have a GUI. GUITAR automates GUI testing by using model-based techniques. An overview of the model used by GUITAR is presented in Section 4.

## 3 Process Support for the Testing Loops

Users interact with a GUI by performing *events* on some widgets, such as clicking on a button, opening a menu, and dragging an icon. During GUI testing, test cases, consisting of sequences of events are executed on the GUI.[6] As discussed in Section 2, the most popular tools used to develop GUI test cases are capture/replay tools, which are largely manual. Our experience with GUI testing shows that a tester cannot use these tools to develop a test suite that covers a significant portion of the GUI (an extremely resource-intensive task) for continuous testing. Test cases obtained from capture/replay tools are very fragile and most of them would become unusable after a few GUI modifications [9]. Test cases become unusable for the modified GUI either because the input event sequence can no longer execute on the GUI or because the expected output (*i.e.*, in the form of assertions) stored with the test case becomes obsolete. During continuous testing, the testers will have to keep updating the GUI test suite, develop new test cases, and delete obsolete ones. These activities are extremely resource intensive, especially with the code churn rates observed in popular OSS.

The only practical alternative is to use model-based techniques to generate and maintain test cases automatically during OSS evolution. The loop-based technique employs GUI models to generate test cases. Three types of concurrently executing processes support the continuous GUI testing technique. Several instances of these processes may

[3] `http://www.dre.vanderbilt.edu/scoreboard/`

[4] `http://mercuryinteractive.com`

[5] `http://junit.org/news/extension/gui/index.htm`

[6] We have shown in earlier work that simply executing each event in isolation is not enough for effective GUI testing [11].

be active at any time. The most frequently executing process that supports crash testing is essentially a two-stage code commit with an automated GUI testing intervention step. A developer who has made a change to a part of the GUI code "checks-in" the changes. An instance of the crash testing process is automatically launched at the server that hosts the code repository (in general, this could be a dedicated computer that is linked to the repository server). A reverse engineering technique [10] is used to automatically obtain a model of the GUI. This model is used to generate crash test cases, which are then automatically executed on the newly modified GUI. "Software crashes" are reported back to the specific developer who checked-in the changes along with the test cases that caused the crash. The developer debugs the GUI and resubmits the changes. Only the previously failed test cases are re-executed; if they pass, the code changes are made permanent in the repository. If two or more developers are modifying different parts of the GUI, multiple instances of crash testing are created, one for each check-in. This ensures that faulty changes do not interact. Note that this process does not require any manual intervention, it is very fast, and gives very specific type of feedback to the developer involved, *i.e.*, whether the software crashed or not.

Every night, a process that supports smoke testing is launched to ensure that changes made to the GUI during a 24 hour period (this interval length is tunable) are integrated properly. The smoke testing process is launched automatically; it employs the reverse engineering technique to obtain the GUI model, which is used for test case generation. The goal of this testing process is to conduct reference testing, using the previously tested version as a baseline. Hence, the previous version is used as a *test oracle* (a mechanism that determines whether a software being tested is executing correctly). As test cases are executed automatically on the latest GUI version, its state after each event is compared to the baseline and mismatches are reported. Although the process described thus far is fully automatic, the mismatches (that are reported to all developers involved in the latest changes) need to be examined manually to weed out false positives. False positives are expected to exist since the software has been modified, leading to expected changes between the new and baseline version.

Periodically, the developers may want to develop and execute a more thorough test suite that looks for errors beyond crashes and differences between the latest and previous versions. We call this process *comprehensive testing*. In this case, the developers create test cases (perhaps by using capture/replay tools) and test oracles to determine if the GUI is executing correctly. In our work to date, we have used automated techniques to do comprehensive GUI testing. For test case generation we have used AI Planning [11]; for test oracle creation, we have used pre- and postconditions [9].

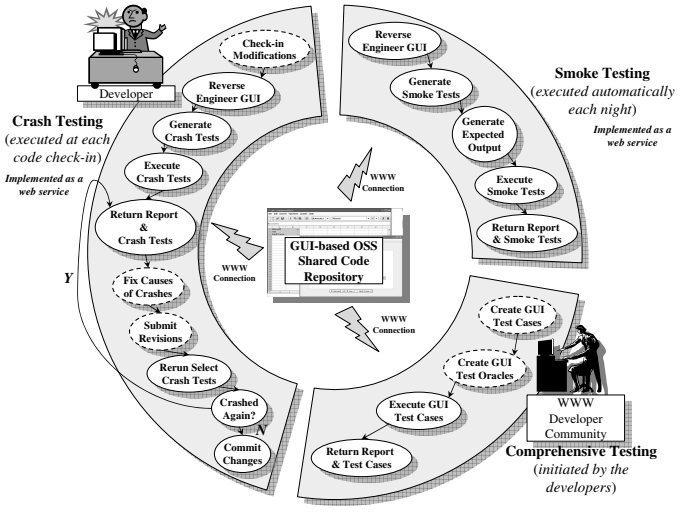


**Figure 1. Continuous GUI Testing.**

Once the test cases have been obtained, they are replayed automatically on the GUI and errors are reported to all the developers.

The processes described above are summarized in Figure 1. The dashed ovals are the activities that are performed by the developer, and are hence resource intensive. Other activities are done automatically by our tools. In all cases, we develop the test cases and test oracles automatically using model-based techniques. The key reason for our success is a 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) [7] described next.

## 4 High-Level Overview of GUI Model

There are two parts of the GUI model, one used for test case generation and the other for test oracle creation. We now describe each of these parts. Note that to conserve space, we provide a high-level overview needed to understand the basic concepts; details have been published in earlier reported work [7, 11, 12, 19].

Intuitively, an event-flow graph (EFG) model represents 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 a path through the EFG for the `Main` and `Replace` windows of the MS NotePad software is shown in Figure 2(a). The path contains six events: <`Edit`,[7] `Replace`, `Type-in-text`, `Find Next`, `View`, `Status Bar`>. Since this path is modeled in the EFG, a MS NotePad user can execute it.

[7] Actually, the event is `Click-on-Edit`; for brevity, we will refer to events by their associated widget names.

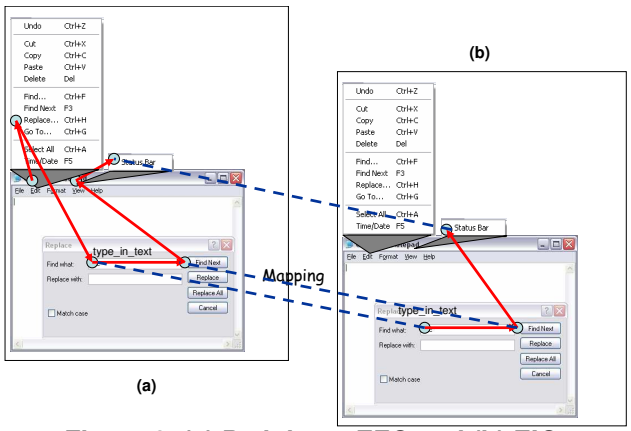


**Figure 2. (a) Path in an EFG and (b) EIG.**

Since we know (from personal experience) that the event `Replace` cannot be executed immediately after the event `Cancel`, the edge (`Cancel`, `Replace`) will not be represented in the EFG. An EFG simply represents all *executable* paths of the software.

An EFG models *all possible* (an extremely large number of) event sequences that may be executed on the GUI. Not all these sequences are necessary to test interactions between loosely-coupled parts of an OSS. Abstractions are used to model only specialized (and hence a smaller number of) event sequences. In a typical GUI, 20-25% of the events are used to manipulate the structure of the GUI; examples include events that open/close windows/menus. For example, in Microsoft Word, of the total 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 usually generated automatically by visual GUI-building tools. This code is very unlikely to interact with code for other events; hence very few integration errors are revealed by executing interactions between these events. The remaining events in the GUI are *non-structural events* that do not cause structural changes to the GUI; rather they are used to perform some action; a common example is the `Copy` event used for copying objects to the clipboard.

Events that interact with the underlying software include non-structural events and those that close windows. These events are called system-interaction events. Interactions between these types of events are represented by event-interaction graphs, which are used to generate test cases. Our test cases consist of those event-flow-paths that start and end with system-interaction events, without any intermediate system-interaction events. An event-flow-path is *interaction-free* iff it contains no system-interaction events except the first and last. 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. Note that an event may interact-with itself. Also note that "$e_x$ interacts-with $e_y$" does not necessarily imply that "$e_y$ interacts-with $e_x$." The interact relationship is used to create the event-interaction graph. This graph contains nodes, one for each system-interaction event in the GUI. An edge from node $n_x$ (that represents $e_x$) to node $n_y$ (that represents $e_y$) means that $e_x$ interacts with $e_y$. Of the events shown along the path in Figure 2(a), the interacts-with relationship holds between `Type-in-text` and `Find Next`; and between `Find Next` and `Status Bar`. These relationships are shown as directed edges in Figure 2(b). Mapping between EFG and EIG is shown as dashed lines.

**Test-case Generation:** An event-interaction graph (EIG) may be used in a number of ways to generate sequences of system-interaction events, which form the GUI test cases. For example, event sequences that cover all nodes in the EIG may be generated by enumerating each node. Similarly, event sequences that cover all edges in the EIG may be generated by enumerating each node with its adjacent node. These test cases form the test suite for crash and smoke testing. The remaining question is how to execute the generated event sequences. At execution time, other events needed to "reach" the system-interaction events are generated on-the-fly. A mapping, shown as dashed lines in Figure 2, between EFG and EIG nodes is used to obtain these events. Hence the two sequences <`Type-in-text`, `Find Next`> and < `Find Next`, `Status Bar`> will expand to <`Edit`, `Replace`, `Type-in-text`, `Find Next`> and < `Find Next`, `View`, `Status Bar`> respectively during test-case execution. Additional events needed to "reach" the first event in the test case are also automatically generated.

The test cases exhibit the following properties.

1. The test cases are short; they can be generated and executed very quickly.

2. The test case consists of system-interaction events; changes to the GUI layout, such as moving events from one window to another and changing the menu structure, leave most of the test cases unaffected. Other events are generated on-the-fly *during* test execution. *i.e.*, the path to get to the system-interaction event is generated dynamically.

3. The expected state is stored only for system-interaction events; it will become obsolete only if a system-interaction event is modified.

4. All system-interaction events are executed; most of the GUI's functionality is covered.

5. Each test case is independent and the suite can be distributed.

The most important property of these tests is that they can be generated and executed automatically using GUITAR. Our GUITAR system contains several modules that makes this automation possible. We discuss these modules in Section 5.

**Test Oracle Creation:** The second part of the model is used to create test oracles, which determine whether a software executed as expected [2]. The test oracle may either be automated or manual; in both cases, the actual output is compared to a presumably correct expected output.

• **Oracles for crash tests:** The test cases defined earlier (*i.e.*, those that cover all edges of an EIG) 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. Crash tests use the detection of crashes as test oracles.

• **Oracles for smoke tests:** Smoke tests use automated test oracles that enable the detection of other GUI problems that may not necessarily be manifested as a software crash. Since the goal of smoke testing is to ensure that the software has not "broken" during modifications, automated GUI test oracles check that the "software does what it was doing before modifications" were made. During smoke testing, the modified GUI version is compared with the original version and changes are reported. The steps involved are: (1) execute the test case on the original GUI and collect state information, (2) execute the same test cases on the modified version, and (3) compare the GUI's state with the stored information and report any changes.

• **Oracles for comprehensive testing:** A specifications-based approach is used to create test oracles for the comprehensive test cases. The process involves developing formal specifications for each event in the form of preconditions (the partial state of the GUI in which the event can execute, *e.g.*, a button is enabled) and effects (the changes to the GUI state after the event has executed, *e.g.*, a window is closed). Details of this approach were described in earlier reported work [9].

## 5 Experiment

We now present an experiment to determine whether popular GUI-based OSS, developed by a community of developers connected via the WWW, have faults that may be detected using our approach. More specifically, we are interested in answering the questions: (1) Do popular web-based community-driven GUI-based OSS have problems that can be detected by our automated techniques? (2) Do these problems persist across multiple versions of the OSS?

To answer these questions, we conduct an experiment using several popular web-based community-driven GUI-based OSS downloaded from SourceForge.net. We then execute our fully-automatic crash testing process on them and report problems. We also download previous versions of these applications and determine how long these problems have been in the code. Note that we tested versions that the developer community chose to make available online. We expect that these applications have undergone some QA before release.

**Implementation:** We have implemented all the modules of GUITAR. Using these modules, GUITAR is able to automatically analyze the GUI, create event-flow and event-interaction graphs, generate test cases and test oracles, and execute the test cases on an instrumented GUI. Coverage reports are created automatically. As the GUI evolves, GUITAR updates its test suite and test oracles automatically. The *GUI ripper* is the automated module that creates the event-flow graphs, which are then converted to event-interaction graphs. "GUI Ripping" is a dynamic process in which the software's GUI is automatically "traversed" by opening all its windows and extracting all their widgets (GUI objects), properties, and values. The *test-case generator* uses the event-interaction graphs to create the test cases. The *test-oracle* generator automatically executes the generated test cases on the latest GUI version and stores the captured state. Coverage evaluation serves as a useful guide to additional testing, whether it is done for the next build or for future comprehensive testing. In GUITAR, we evaluate conventional code coverage in terms of statements, branches, methods, classes, packages, and files. To collect the coverage information, we use source-level *code instrumenters*. The *test executor* is capable of executing an entire test suite automatically on the GUI. It performs all the events in each test case and invokes the test oracle to compare the actual output with the expected output. If the event requires text input, then the values are read from a database, initialized by the tester. 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.

**Subject Applications:** We chose the following four applications with GUIs developed using Java Swing:

1. **FreeMind**[8], which is a premier free mind-mapping[9] software written in Java. It has an all time activity of 99.72%. We tested versions 0.0.2, 0.1.0, 0.4, 0.7.1, 0.8.0RC5 and 0.8.0.

2. **GanttProject**[10], which is a project scheduling application written in Java and featuring Gantt chart, resource management, calendaring, import/export (MS Project, HTML, PDF, spreadsheets). It has an all time activity of 98.12%. We tested versions 1.6, 1.9.11, 1.10.3, 1.11, 1.11.1, and 2.pre1.

3. **JMSN**[11], which is a pure Java Microsoft MSN Messenger clone, including Instant messaging, File Send/Receive, msnlib (for developers), and additional chat log, etc. It has an all time activity of 98.93%. We tested versions 0.9a, 0.9.2, 0.9.5, 0.9.7, 0.9.8b7, and 0.9.9b1.

[8] http://sourceforge.net/projects/freemind
[9] http://en.wikipedia.org/wiki/Mind_map
[10] http://sourceforge.net/projects/ganttproject
[11] http://sourceforge.net/projects/jmsn

4. **CrosswordSage**[12], which is a tool for creating (and solving) professional looking crosswords with powerful word suggestion capabilities. When tested, it had an activity percentile (last week) of 98.21%. We tested versions 0.1, 0.2, 0.3.0, 0.3.1, 0.3.2, and 0.3.5.

The first three of the above applications were chosen due to their popularity, active community of developers, and high all-time activity. Crossword Sage was chosen since it is fairly new (it was registered in mid-Sep. 2005) with several versions. We tested all the above applications on the Windows 2000 Professional platform. GUITAR-setup included setting up a database for text-field values. Since we wanted our overall process to be fully automatic, we used a "default" database that contains one instance for each of the text types in the set {*negative number, real number, long file name, empty string, special characters, zero, existing file name, non-existent file name*}. Note that if a text field is encountered in the GUI (represented as an event called `type-in-text`), one instance for each text type is tried in succession. We also setup the test oracle to detect crashes – for these applications, we define a crash as an uncaught exception thrown during test case execution.

The overall process executed without any human intervention in 5-8 hours; one machine per application. The reverse engineering, model creation, test case generation steps took 2-3 minutes per application. The test cases execution took the remaining time. Note that we could have greatly speeded up test case execution by splitting up the test suite for each application across multiple machines. As noted earlier, the test cases are independent.

### 5.1 Results

Our crash testing process produced several interesting results, which we now present. We first manually examined all the crash logs and identified the event sequences that caused the crash. The results of our analysis are summarized next. Note that version numbers are shown in parenthesis. Each listed bug will be referred by its *bug number* in later discussions.

**FreeMind:** 1. NullPointerException when trying to open a non-existent file (0.0.2, 0.1.0);

2. FileNotFoundException when trying to save a file with a very long file name (0.0.2, 0.1.0, 0.4);

3. NullPointerException when clicking on some buttons on the main toolbar when no file is open (0.1.0);

4. NullPointerException when clicking on some menu items if no file is open (0.1.0, 0.4, 0.7.1, 0.8.0RC5);

5. NullPointerException when trying to save a "blank" file (0.1.0);

6. NullPointerException when adding a new node after toggling folded node (0.4);

7. FileNotFoundException when trying to import a non-existent file (0.4, 0.7.1, 0.8.0RC5, 0.8.0);

8. FileNotFoundException when trying to export a file with a very long file name (0.7.1, 0.8.0RC5, 0.8.0);

9. NullPointerException when trying to split a node in "Edit a long node" window (0.7.1, 0.8.0RC5, 0.8.0);

10. NumberFormatException when setting non-numeric input while expecting a number in "preferences setting" window (0.8.0RC5, 0.8.0);

**Gantt Project:** 1. NumberFormatException when setting non-numeric inputs while expecting a number in "New task" window (1.6);

2. FileNotFoundException when trying to open a non-existent file (1.6);

3. FileNotFoundException when trying to save a file with a very long file name (1.6, 1.9.11, 1.10.3, 1.11, 1.11.1, 2.pre1);

4. NullPointerException after confirming any preferences setting (1.9.11);

5. NullPointerException when trying to save the content to a server (1.9.11);

6. NullPointerException when trying to import a non-existent file (1.9.11, 1.10.3, 1.11, 1.11.1, 2.pre1);

7. InterruptedException when trying to open a new window (1.10.3);

8. Runtime error when trying to send e-mail (1.11, 1.11.1, 2.pre1);

**JMSN:** 1. InvocationTargetException when trying to refresh the buddy list (0.9a, 0.9.2);

2. FileNotFoundException when trying to submit a bug/request report because the submission page doesn't exist (0.9a, 0.9.2, 0.9.5, 0.9.7, 0.9.8b7, 0.9.9b2);

3. NullPointerException when trying to check the validity of the login data (0.9.7, 0.9.8b7, 0.9.9b2);

4. SocketException and NullPointerException when stopping a socket that has been started (0.9.8b7, 0.9.9b2);

**Crossword Sage:** 1. NullPointerException in Crossword Builder when trying to delete a word (0.3.0, 0.3.1);

2. NullPointerException in Crossword Builder when trying to suggest a new word (0.3.0, 0.3.1, 0.3.2, 0.3.5);

3. NullPointerException in Crossword Builder when trying to write a clue for a word (0.3.0, 0.3.1, 0.3.2, 0.3.5);

4. NullPointerException when loading a new crossword file (0.3.5);

5. NullPointerException when splitting a word (0.3.5);

6. NullPointerException when publishing the crossword (0.3.5);

From the above list of severe problems, we see that fielded GUI-based OSS developed by a community of developers have problems that are quickly uncovered using our GUI-integration testing process. Since the overall process is completely automatic, crash testing, integrated with

[12]http://sourceforge.net/projects/crosswordsage

CVS, can discover these problems before they are found by users.

We were surprised to find that some bugs existed across applications. This was due to shared open-source GUI components. For example, Bug#2 in FreeMind and Bug#3 in GanttProject are identical since both these applications share a *FileSave* component. This component throws a FileNotFoundException when given a very long file name, which cannot be handled by the Windows operating system. This particular bug does not show up after Version 0.4 of FreeMind; however, the same bug still shows up when the user tries to *export* a file with a very long file name. This observation shows that OSS that use shared components must "sanitize" inputs before passing them to the shared components.

Figure 3 gives an overview of bug history across versions of each application. The x-axis represents the versions; the y-axis uses the bug numbers assigned earlier. Each bug that led to one crash is represented by a small filled circle; bugs that led to multiple crashes are represented by an asterisk. If the same bug persisted across multiple versions, the circles (or asterisks) are connected by a horizontal line. For example, many crashes are caused by Bug#3 in FreeMind (several toolbar buttons should be disabled if there is no file opened).

From Figure 3, we observe that many bugs are persistent across versions. For example, Bug#4, #7, #8, #9 and #10 in FreeMind persisted across several versions before they were discovered and fixed. The same observation holds for the other applications. In fact, Bug#3 in GanttProject appeared in the first version we tested (we chose Version 1.6 since it is the first version with default language English); it exists in all versions, including the latest version.

Since SourceForge has a bug reporting/tracking tool for each project, we reported some bugs. For example, we reported Bug#4 in FreeMind for version 0.8.0RC5 (bug #1245216 in SourceForge[13]).

In response to our report, the developers fixed this bug in release 0.8.0. We intend to report all other bugs, especially the ones in the latest versions of all the applications.

Figure 3 leads to another observation that we have found is consistent with our experience with other OSS. There are fewer bugs in the first version than in later versions. For example, there are two crash-causing bugs in Version 0.0.2 of FreeMind. Typically, the first version of an OSS is relatively simple and is developed by a small group of core developers. This version typically undergoes QA before its first release; hence it is reasonably stable. Versions 0.1.0 and 0.2.0 of CrosswordSage have no bugs because they are very simple. The only change that was made from Version 0.1.0 to Version 0.2.0 was a new help document. As the developer community grows, the application becomes more complex and prone to bugs. For example, Bug#10 in FreeMind was first introduced when a new "preference setting" functionality was added. Similarly, there was a new feature added to Version 0.3.0 of Crossword Sage; this new feature introduced some bugs that we detected. There were more features added in Version 0.3.5; bugs were detected in the added part of code.

By default, we tested all our applications in one machine configuration on Windows 2000 Professional. We observed that altering this "default" configuration helps to uncover more bugs. In a preliminary study, we tested GanttProject in a new configuration with a much lower memory setting than the default configuration. We found that Bug#4 and Bug#7 surface only in this low memory configuration. In case of Bug#4, the application tries to repaint all the GUI windows/widgets after the preferences setting have changed; in low memory, this causes a substantial delay for the user. Any event performed during the slow repainting process causes an uncaught NullPointerException exception. In case of Bug#7, the application requires additional time to open new windows; if a user performs a new event during this time, the result is an uncaught InterruptedException exception.

We now describe some of the reasons for these crashes. We identified four reasons: (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 GanttProject and Freemind expect an integer input; providing a string resulted in a crash. In some instances, a "very long" text input also resulted in a crash, such as providing a "very long" text input as the file name while saving such a file sometimes leads to FileNotFoundException. (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 these open-source 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. (4) *Obsolete external resources.* Some of the crashes in JMSN were caused by the test cases that were trying to retrieve information from a web page that is no longer available.

Note that in the context of a large development commu-

[13] http://sourceforge.net/tracker/index.php?func=detail&aid=1245216&group_id=7118&atid=107118

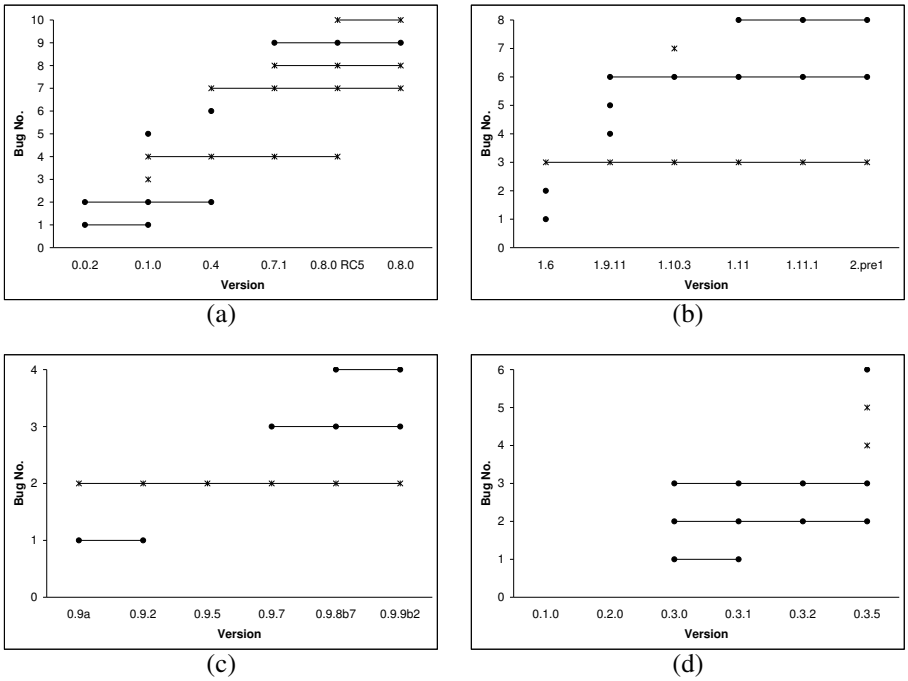


**Figure 3. Bug History Over Versions.**

nity, it is important to focus feedback to relevant developers who can quickly fix the newly introduced bugs. Our crash test reports are concise (they contain the crash log and test cases) and are sent only to the developer who initiated the code check-in. Together with the above "common causes of crashes," the developer can debug the new code and resubmit the changes. This technique ensures that all developers will not get overwhelmed with too many error messages each time a change is checked-in.

The above results of crash testing, together with our previously published work on smoke and comprehensive GUI testing demonstrates that GUI integration testing is useful, especially for quick integration testing of evolving GUI OSS that is being developed by a community of developers interconnected on the WWW.

## 6 Conclusions & Future Work

This paper presented a new approach for continuous integration testing of web-based community-driven GUI-based OSS. Since GUI testing is extremely resource intensive, the approach implicitly distributes the testing tasks by leveraging the developer community and targeting feedback to specific sub-groups of developers. Three concentric loops with associated processes were presented. The innermost loop called crash testing operates on each code check-in of the GUI software and performs a quick-and-dirty, fully automatic integration test of the GUI software; feedback is directed to the developer who initiated the check-in. The second loop called smoke testing operates on each day's GUI build and performs functional reference testing of the newly integrated version of the GUI; developers who changed the OSS since the last smoke test run are given feedback. The third (outermost) loop called comprehensive GUI testing conducts detailed integration testing of a major GUI release. Our earlier work showed that comprehensive and smoke testing are useful. This paper evaluated the crash testing process in an experiment involving four popular OSS. The experiment showed that (1) the GUI-based testing approach helps to find integration problems in fielded GUI-based OSS, (2) several problems persist across multiple versions of OSS, (3) errors surface in different OSS that share problematic open-source GUI components, and (4) the first version (likely created by a core group of developers) of most OSS is relatively stable; problems surface as additional developers add functionality. Post-experiment analysis revealed that most of these problems are caused by incorrect integration of different parts of the OSS.

There are several new directions for additional research. Short-term future work will focus on a more detailed study of the overall benefits of this technique. Students of the undergraduate Software Engineering class at the University of Maryland have been developing a GUI-based OSS called TerpOffice.[14] They will evolve the software in the Spring 2006 semester. They will employ our new technique to automatically test the software incrementally. This will also extend our subject application pool for study.

The applications that we have studied thus far have sev-

[14] http://www.cs.umd.edu/users/atif/TerpOffice/

eral windows with numerous widgets; most of the user events are mouse-clicking events on the widgets. Values are read from a database and automatically entered into text-boxes. In the medium-term, we intend to extend our research to applications such as web applications that have complex back-ends. One particular aspect of our results has direct applicability to testing web applications. Our test case re-player uses Java reflection to perform events on GUI widgets. This mechanism also allows us to perform events that are disabled, and hence cannot be performed by end-users. For example, a user cannot change text in a disabled text-box. However, our test case re-player can perform this operation via reflection. As a preliminary study, we generated test cases that allow us to click on disabled buttons and change text in disabled text-boxes. As expected, this led to several uncaught exceptions. For example, we encountered an uncaught NullPointerException in FreeMind when clicking on a disabled button, and a StringIndexOutofBoundsException when setting a wrong string in a text-box that expects date format input. While not immediately relevant to GUI testing, this result has implications for web bypass testing [14], in which a tester bypasses the client user interface (and hence any user-interface constraints) and interacts directly with the web sever via http request events. We envision a new web "crash testing" technique that combines bypass and GUI crash testing.

In the long term, we will study the interaction between our three loops. In particular, we will study whether one loop (say the outermost) can benefit from the execution of the inner loops. We envision that the inner loops may be able to annotate the event-interaction graphs; the outer loops may be able to use these annotations to improve test case generation. For example, event sequences that have already been tested by inner loops may not need to be re-tested by the outer loops. We will also study the need for additional loops.

## Acknowledgments

This work was partially supported by the US National Science Foundation under NSF grant CCF-0447864 and the Office of Naval Research grant N00014-05-1-0421.

## References

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

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

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

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

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

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

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] A. M. Memon and Q. Xie. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *IEEE Transactions on Software Engineering*, 31(10):884–896, Oct. 2005.

[13] B. A. Myers. Why are human-computer interfaces difficult to design and implement? Technical Report CS-93-183, Carnegie Mellon University, School of Computer Science, July 1993.

[14] J. Offutt and W. Xu. Generating test cases for web services using data perturbation. *SIGSOFT Softw. Eng. Notes*, 29(5):1–10, 2004.

[15] 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.

[16] 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.

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

[18] Q. Xie and A. Memon. Designing and comparing automated test oracles for gui-based software applications. *ACM Transactions on Software Testing and Methodology*, to appear.

[19] Q. Xie and A. M. Memon. Rapid crash testing for continuously evolving GUI-based software applications. In *Proceedings of The International Conference on Software Maintenance 2005 (ICSM'05)*, pages 473–482, Budapest, Hungary, Sept. 2005.