Automatically Repairing Event Sequence-Based GUI Test Suites for Regression Testing

Atif M. Memon
4115 A. V. Williams Building, Department of Computer Science
University of Maryland, College Park, MD 20742, USA

Although graphical user interfaces (GUIs) constitute a large part of the software being developed today and are typically created using rapid prototyping, there are no effective regression testing techniques for GUIs. The needs of GUI regression testing differ from those of traditional software. When the structure of a GUI is modified, test cases from the original GUI's suite are either reusable or unusable on the modified GUI. Because GUI test case generation is expensive, our goal is to make the unusable test cases usable, thereby helping to retain the suite's event coverage. The idea of reusing these unusable (*obsolete*) test cases has not been explored before. This paper shows that a large number of test cases become unusable for GUIs. It presents a new GUI regression testing technique that first automatically determines the usable and unusable test cases from a test suite after a GUI modification, then determines the unusable test cases that can be repaired so that they can execute on the modified GUI, and finally uses *repairing transformations* to repair the test cases. This regression testing technique along with four repairing transformations has been implemented. An empirical study for four open-source applications demonstrates that (1) this approach is effective in that many of the test cases can be repaired, and is practical in terms of its time performance, (2) certain types of test cases are more prone to becoming unusable, and (3) certain types of "dominator" events, when modified, make a large number of test cases unusable.

Categories and Subject Descriptors: D.2.5 [**Testing and Debugging**]: Regression Testing; K.6.3 [**Software Management**]: Software maintenance

General Terms: Verification, Reliability

Additional Key Words and Phrases: Graphical user interfaces, regression testing, test maintenance, repairing test cases, test case management

1. INTRODUCTION

Graphical User Interfaces (GUIs) are pervasive in today's software systems and constitute as much as half of software code [Myers 1995; Memon 2001]. The correctness of a software system's GUI is paramount in ensuring the correct operation of the overall software system [Kepple 1994; Ostrand et al. 1998]. One way, and the common way to gain confidence in a GUI's correctness is through comprehensive testing. GUI testing requires that test suites (containing test cases – sequences of

Author's address: Atif M. Memon, 4115 A. V. Williams Building, Department of Computer Science, University of Maryland, College Park, MD 20742, USA

GUI *events* that exercise GUI *widgets*) be developed and executed on the *application under test* (AUT) [Walworth 1997; Kepple 1992; Memon 2002]. However, currently available techniques for obtaining GUI test suites are resource intensive, requiring significant human intervention. Even though several automated approaches have been proposed [Shehady and Siewiorek 1997; Memon et al. 2001; Memon and Xie 2005; Memon et al. 2005; White and Almezen 2000; Kasik and George 1996], in practice, GUI test suites are still being developed manually using *capture/replay tools* [Hicinbothom and Zachary 1993]. Moreover, most GUIs are designed using *rapid prototyping* [Myers 1995; Wittel, Jr. and Lewis 1991; Rosenberg 1993], in which software is modified and tested on a continuous basis. The continuous modification of a GUI requires that most of the test cases in the suite be reusable across versions, as it is expensive to develop a new suite for each version.

When a GUI is modified, the test cases in a test suite fall into one of two categories: usable and unusable. In the "*usable*" category, the test cases can be rerun on the modified GUI. In the "*unusable*" category, the test cases cannot be rerun to completion. For example, a test case may specify clicking on a button that may have been deleted or moved. Similarly, test cases may also become unusable because of GUI layout changes such as the creation of a new menu hierarchy, moving a widget from one menu to another, and moving a widget from one window to another. The unusable test cases are identified only after they have been executed, leading to severe waste of valuable resources.

An earlier report of this research presented a new regression testing technique that helps to retain a test suite's event coverage by reusing existing test cases from the original GUI's suite that have become unusable for the modified GUI by automatically *repairing* them so that they can execute on the modified GUI [Memon and Soffa 2003]. With this repairing technique, a tester can rerun test cases that are usable for the modified GUI, as currently done, repair and rerun previously unusable test cases, and create new test cases to ensure adequate event coverage for the resulting suite. The repairing technique leverages existing GUI representations developed for a GUI testing framework [Memon et al. 2001; Memon et al. 2000; Memon 2001]. A model of the event structure of a GUI is created and used to determine the modifications; it is then used to check whether each existing test is usable on the modified GUI and if not, repair it if possible using one of several user-defined *repairing transformations*.

This paper empirically assesses, via a study on multiple versions of four open-source software subjects, the technique presented in the earlier work [Memon and Soffa 2003]. The results of the study show that (1) a large number of test cases in fact become unusable for modified GUI versions, (2) the repairing algorithms, with four simple transformations, are able to repair more than half of these test cases, (3) short test cases are less likely to become unusable, and (4) changes to certain "dominator" events cause a large number of test cases to become unusable.

The repairing technique is general, in that it can be used on test suites that have been developed using any method, *e.g.*, capture/replay, state-machines [White and Almezen 2000], event-flow graphs (EFGs) [Memon et al. 2005; Memon and Xie 2005], AI planning [Memon et al. 2001], and genetic algorithms [Kasik and George 1996]. Tools for automatic creation of the representations, and identification and

repair of unusable test cases have been developed and made available on-line.[1]

More specifically, the contributions of this paper include:

(1) the first regression testing technique that helps to retain a suite's event coverage across software versions by automatically obtaining new test cases from unusable test cases,
(2) a checker that determines if an existing test case is usable or unusable on a modified GUI and, if unusable, determines if it can be repaired,
(3) an extensible repairer that employs user-defined transformations to repair previously unusable test cases,
(4) empirical assessment on 19 versions of real open-source applications that the repairing technique is effective and practical,
(5) integration of the regression testing technique into a framework for GUI testing,
(6) identification of characteristics of GUI changes that lead to unusable test cases, and
(7) identification of characteristics of GUI test cases that make them unusable.

It should be noted that a test case that has an associated *test oracle* (a mechanism that determines whether a test case passed or failed) may become unusable because the test oracle no longer encodes the correct expected behavior of the modified software. Executing a test case with such a test oracle will produce misleading results. "Repairing" or regenerating the test oracle is an important and complex task; it is a subject for future work. The technique described in this paper assumes the availability of "global" test oracles that are not associated with individual test cases; rather they monitor the software during test-case execution, checking for overall contract violations, *e.g.*, software crashes/freezing. Several earlier reports, including our own [Xie and Memon 2006; Yuan and Memon 2007] have demonstrated the usefulness of such "global" test oracles obtained from assertions (embedded in the code) that check the partial state of the program during the execution of a test suite [Rosenblum 1995; Voas 1997] or a list of programmer-specified contracts that must not be violated at any time during its execution [Pacheco et al. 2007].

Structure of the Paper: The next section presents background and related research. A motivating example and algorithms for checking and repairing test cases are presented in Section 3. Section 4 describes the results of an empirical study performed on four open-source applications. Section 5 concludes with a discussion on ongoing and future work.

2. RELATED WORK

Although regression testing research has received a lot of attention [Binkley 1997; Rosenblum and Weyuker 1997; Rothermel and Harrold 1997; 1998], there has been very little reported research on GUI regression testing. The research presented in this paper leverages several existing techniques, which are discussed next.

Regression Testing of Conventional Software: Several strategies for regression testing of conventional software have been proposed and used [Harrold et al.

[1] `http://guitar.cs.umd.edu`

1993; Rosenblum and Rothermel 1997; Kung et al. 1996]. One regression testing strategy proposes rerunning all test cases that still belong to the new version's input domain. Because this *retest-all strategy* is resource intensive, numerous efforts have been made to reduce its cost. *Selective retest techniques* [Agrawal et al. 1993; Benedusi et al. 1988; Harrold and Soffa 1989] attempt to reduce the cost of regression testing by testing only selected parts of the software. These techniques have traditionally focused on two problems: (1) *regression test selection problem*, *i.e.*, selecting a subset of the existing test cases [Rothermel and Harrold 1997], and (2) *coverage identification problem*, *i.e.*, identifying portions of the software that require additional testing [Rothermel and Harrold 1997]. Solutions to the regression test selection problem compare structural representations (*e.g.*, control-flow graphs [Rothermel and Harrold 1997], control-dependence graphs [Rothermel and Harrold 1993]) of the original and modified software. Test cases that cause the execution of different paths in these structures are likely to be selected for retesting. Among selective retest strategies, the *safe approaches* require the selection of every existing test case that exercises any program element that could be affected by a given program change. On the other hand, *minimization approaches* attempt to select the smallest set of test cases necessary to test affected program elements at least once [Rothermel et al. 1998]. Other regression testing techniques include analyzing changes to functions, types, variables, and macro definitions [Rosenblum and Rothermel 1997], using def-use chains [Harrold et al. 1993], constructing procedure dependence graphs [Binkley 1997], and analyzing code and class hierarchy for object-oriented programs [Kung et al. 1996].

There are no reported techniques to reuse test cases that no longer belong to the modified program's input domain. Several authors have recognized the need for the *identification* (not reuse nor repair) of such test cases [Onoma et al. 1998; Beizer 1990]. Onoma *et al.* [Onoma et al. 1998] point out that in practice, test case *revalidation*, which aims to identify test cases that are no longer usable for the modified software is done manually. Test case revalidation requires the tester to examine the AUT's specifications and existing test suite; a test case that is no longer usable is discarded. These activities are manual and can be quite expensive.

GUI Regression Testing: White [White 1996] proposes a Latin square method to reduce the size of a GUI regression test suite. The underlying assumption made therein is that it is enough to check pair-wise interactions between menu-items of the GUI. The technique requires that each menu item appears in at least one test case. This strategy seems promising since it also employs GUI events. However, the technique needs to be extended to GUI items other than menus. Moreover, detailed studies need to be conducted to verify whether the pair-wise interactions checking assumption is sufficient.

White *et al.* [White et al. 2003] have extended their work on complete interactions sequences (CIS) to develop a new firewall [White et al. 2005] regression testing approach for GUIs. A CIS is a state-machine model that partitions the GUI state space into different machines based on user tasks [White and Almezen 2000]. The test designer/expert manually identifies a *responsibility*, *i.e.*, a user task that can be performed with the GUI. For each responsibility, the test designer identifies a machine model called the CIS. The CIS foundation is used to develop a new firewall

method that selects only those GUI objects in a firewall that need be regression tested.

Our own previous work studied the fault-detection effectiveness of GUI test cases for rapidly evolving software [Memon et al. 2005; Memon and Xie 2005]. Some shortcomings of modern smoke regression techniques, such as the inability to automatically retest GUIs are addressed. Various empirical results and solutions to this problem are discussed. More specifically, the requirements for GUI smoke testing are identified and a GUI smoke suite is formally defined as short sequences of events satisfying the *event-interaction* adequacy criterion [Memon et al. 2001]. In addition, the use of the Daily Automated Regression Tester (DART), which is an automated regression testing process, is presented. Finally, the results of empirical studies demonstrate the feasibility of the overall smoke testing process in terms of execution time and storage space. Some other equally important results indicate that smoke tests cannot cover certain parts of the code and that having comprehensive test oracles may balance off the lack of large smoke test suites [Xie and Memon 2007].

GUI Regression Testing Practice: Capture/replay tools [Hammontree et al. 1992] are currently the most popular tools used in practice for GUI testing. Capture/replay tools operate in two modes: *capture* and *replay*. In the *capture* mode, tools such as *CAPBAK* and *TestWorks* [Software Research, Inc., Capture-Replay Tool 2003] record mouse coordinates of the user actions as test cases. In the *replay* mode, the recorded test cases are replayed automatically. The problem with such tools is that, since they store mouse coordinates, test cases break even with the slightest changes (*e.g.*, relocation of a widget by a few pixels) to the GUI layout. Tools such as *Winrunner* [WinRunner 2003], *Abbot* [Abbot 2003], and *Rational Robot* [RationalRobot 2003] overcome this problem by capturing GUI widgets rather than mouse coordinates. Although replay is automated, significant effort is involved in creating the test cases and detecting failures. When these test cases are executed on the modified version of software, they may fail either due to (1) errors in the AUT or (2) GUI modifications. Unless the latter type of failures (also called *false positives*) are manually revalidated before re-testing, they need to be weeded out manually after they have been executed. Both these activities are resource intensive. Additional test cases are developed to replace the test cases that lead to false positives and to test new functionality.

A popular alternative to capture/replay tools used for GUI regression testing in practice is to use test harnesses that "bypass" the GUI and invoke methods of the underlying code or "business logic" as if initiated by a GUI [Thatcher 1994]. Examples of some tools that may be used for such bypass testing include extensions of *JUnit* such as *JFCUnit*, *Abbot*, *Pounder*, and *Jemmy Module* [JUnitResources 2005]. Test cases developed using this approach are more resilient to GUI layout changes. However, this approach not only requires major changes to the software architecture (*e.g.*, keep the GUI software "light" and code all "important" decisions in the business logic [Marick 2002]), it also does not perform testing of the end-user software.

Although none of the above techniques address the problem of reusing unusable test cases, several of them provide the necessary foundation for this work. For

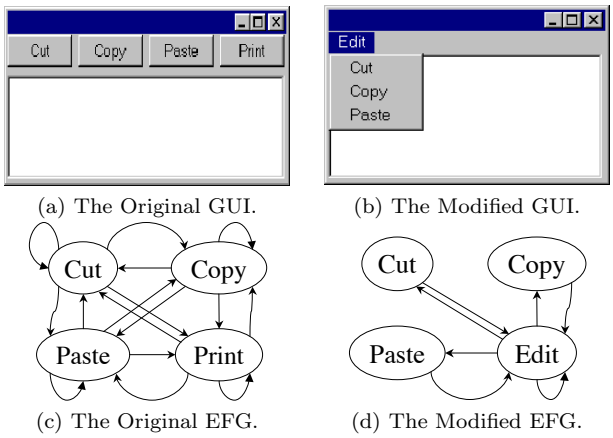
example, the idea of comparing structures (control-flow and program-flow graphs) is leveraged here, *i.e.*, EFGs of the original and modified GUIs are compared to determine modifications. These modifications are used to identify unusable test cases (obtained via capture/replay tools) and to repair them. The scope of this research is restricted to test cases that are made unusable due to structural changes in a GUI. If the semantics of an event change, then the test cases are considered to be usable even though the GUI's output may change.

3. THE OVERALL REGRESSION TESTING METHOD

This section presents an overview of the new regression testing technique. Because details (algorithms) have been presented in earlier work [Memon and Soffa 2003], and due to lack of space, the focus here is to present an overview needed to understand the empirical study of Section 4.

3.1 A Motivating Example

We first start with examples of GUI modifications, and test cases that have become unusable for the modified GUI. We then use these examples to present an intuitive idea of how analysis of GUI changes can help to identify unusable test cases, and how unusable test cases may be repaired to obtain new test cases.



(a) The Original GUI. (b) The Modified GUI.

(c) The Original EFG. (d) The Modified EFG.

Fig. 1. A Regression Testing Example.

Figure 1(a) and (b) present a GUI and its modified version. The original GUI consists of four events, `Cut, Copy, Paste`, and `Print`, all directly accessible when the GUI is invoked. The modified GUI contains three of the four original events; `Print` has been deleted and the remaining 3 events have been grouped into a pull-down menu, which is opened by clicking on `Edit`. The semantics of individual events and the underlying code have not changed. Figure 1(c) shows the EFG of the GUI of Figure 1(a). In this research, the EFG model is used extensively to automate regression testing. EFGs have been discussed in detail in previously reported work [Memon et al. 2001; Memon and Xie 2005; Memon et al. 2005] – an intuitive overview is given here.

#	Event Sequence	Events Used	Edges Covered
1	`Copy; Print; Cut`	{`Copy, Cut, Print`}	{`(Copy, Print), (Print, Cut)`}
2	`Cut`	{`Cut`}	{}
3	`Cut; Paste`	{`Cut, Paste`}	{`(Cut, Paste)`}
4	`Copy; Cut; Paste`	{`Cut, Copy, Paste`}	{`(Copy, Cut), (Cut, Paste)`}

Table I. Four Event Sequences for the Original GUI.

An EFG models all possible event sequences that may be executed on a GUI. It is a directed graph that contains vertices (that represent events) and edges that represent a relationship between events. An edge from vertex v_x to vertex v_y means that the event represented by v_y may be performed *immediately after* the event represented by v_x. This relationship is called `follows`. Note that a state machine model that is equivalent to this graph can also be constructed – the state would capture the possible events that can be executed on the GUI at any instant; transitions cause state changes whenever the number and type of available events change.

In Figure 1(c), the event `Copy follows Cut`, represented by a directed edge from the vertex labeled `Cut` to `Copy`. In fact, the original GUI's EFG is fully connected with four vertices representing the four events. The modified GUI's EFG is quite different from that of the original GUI; it is no longer fully connected and `Edit` must be performed before any other event can be performed. Each EFG is represented by two sets: (1) a set of vertices $\mathbf{V}$ representing events in the GUI and (2) a set $\mathbf{E}$ of ordered pairs (e_x, e_y), where $\{e_x, e_y\} \subseteq \mathbf{V}$, representing the directed edges in the EFG; $(e_x, e_y) \in \mathbf{E}$ iff e_y `follows` e_x.

The following sets of changes may be obtained, summarizing the differences between the EFGs of the original and modified software:

(1) **events_deleted** = {`Print`}.

(2) **efg_edges_deleted** = { `(Cut, Cut)`, `(Copy, Copy)`, `(Paste, Paste)`, `(Print, Print)`, `(Cut, Copy)`, `(Cut, Paste)`, `(Cut, Print)`, `(Copy, Cut)`, `(Copy, Paste)`, `(Copy, Print)`, `(Print, Cut)`, `(Print, Copy)`, `(Print, Paste)`, `(Paste, Cut)`, `(Paste, Copy)`, `(Paste, Print)`}.

Four event sequences used to test the original GUI are shown in Table I. Column 1 shows the event sequence number, Column 2 shows the event sequence, Column 3 shows the events in the EFG used by the sequence (note that all the GUI events are covered by this small suite), and Column 4 shows the edges of the EFG covered by the test case. The following observations can be made by examining these test cases and the sets computed above:

(1) Because `Print` was deleted from the GUI (**events_deleted**), event sequence 1 is illegal for the modified GUI.

(2) Because `(Cut, Paste)` and `(Copy, Cut)` have been deleted from the GUI (**efg_edges_deleted**), event sequences 3 and 4 are illegal for the modified GUI.

(3) Event sequence 2 is still legal since `Cut` is available in the modified GUI (starting in an initial state in which `Edit` has been performed).

Looking at the original and modified GUIs, event sequences 3 and 4 may be modified (by applying repairing transformations) to obtain legal event sequences. For example, a repairing transformation may insert an event in the test case to make it usable. One application of this repairing transformation to event sequence 3 yields <`Cut; Edit; Paste`> and two applications to event sequence 4 yields <`Copy; Edit; Cut; Edit; Paste`>. These two repaired event sequences are legal and may be used to test the modified GUI; both must be executed in a state of the GUI in which the event `Edit` has been performed already. It is not obvious how event sequence 1 may be repaired as it contains an event, namely `Print`, that is no longer available in the modified GUI; this event sequence may be discarded as non-repairable and not used for regression testing. This example shows that some unusable test cases may not be repairable for a set of transformations. The test designer may later develop a new transformation to repair this test case. After repairing sequences 3 and 4, the test designer can choose from a total of three event sequences and use them for regression testing. Note that this new test suite also covers all the events in the modified GUI. As event sequence 2 has already been executed on the original GUI, and *if* none of the events in this sequence have been modified, the test designer may choose to not rerun it (unless something has changed in the underlying code). In that case, the remaining two event sequences, 3 and 4, can be used for regression testing in addition to any new test cases.

3.2 Usable, Unusable, and Repairable GUI Test Cases

We now present an overview of a model of GUIs that was developed earlier for a GUI testing framework [Memon et al. 2001; Memon et al. 2000; Memon 2001] and use it to formally define unusable, usable, and repairable test cases. In our earlier work, a GUI is modeled as a set of *objects/widgets* $O = \{o_1, o_2, \ldots, o_m\}$ (*e.g.*, `label`, `form`, `button`, `text`) and a set of *properties* $P = \{p_1, p_2, \ldots, p_l\}$ of those objects (*e.g.*, `font`, `caption`). Each GUI will use certain types of objects with associated properties; at any specific point in time, the state of the GUI can be described in terms of the set P of all the properties of all the objects O that the GUI contains. A set of states S_I is called the *valid initial state set* for a particular GUI iff the GUI may be in any state $S_i \in S_I$ when it is first invoked. Of importance to testers are sequences that are permitted by the structure of the GUI. A *legal event sequence* of a GUI is $e_1; e_2; e_3; \ldots; e_n$ where e_{i+1} `follows` e_i. An event sequence that is not legal is called an *illegal* event sequence. For example, in MS Word, `Cut` (in the `Edit` menu) cannot be performed immediately after `Open` (in the `File` menu), and thus the event sequence <`Open`, `Cut`> is illegal (ignoring keyboard shortcuts). Finally, a GUI test case T is a pair $(S_0, e_1; e_2; \ldots; e_n)$, consisting of a state $S_0 \in S_I$, called the *initial state for* T, and a legal event sequence $e_1; e_2; \ldots; e_n$.

If the initial state specified in the test case is no longer reachable in the modified GUI and/or its event sequence has become illegal, then the test case is no longer executable. GUI test case $(S_0, e_1; e_2; \ldots; e_n)$ is *unusable* if a modification of a GUI causes the state S_0 to not be reachable in the GUI or if the sequence $e_1; e_2; \ldots; e_n$ cannot execute to completion.

Unusable test cases cannot be executed on the GUI and are usually discarded. An unusable test case is *repairable* if its initial state S_0 is reachable and its event sequence can be made legal, via repairing transformations, for the modified GUI.

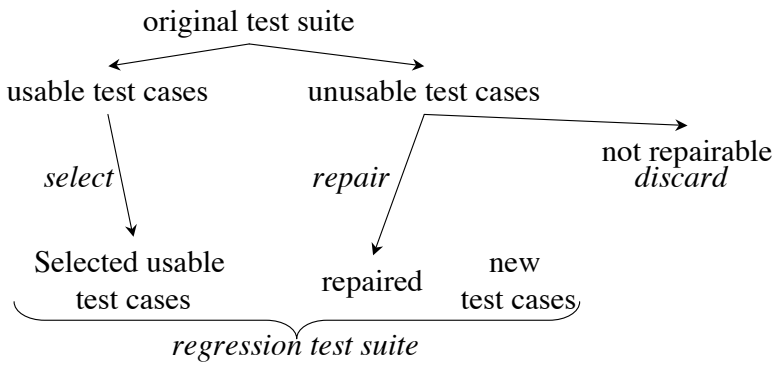


Fig. 2. The New Regression Testing Method.

Four examples of these transformations are discussed in Section 3.3.2.

3.3 Developing the Regression Tester

Our regression testing technique consists of two parts: a checker that categorizes a test case as being usable or unusable; if unusable, it also determines if the test case can be repaired. The second part is the repairer that repairs the unusable, repairable test case. Although for ease of explanation, these two parts are treated individually, they may be merged together in an implementation. The regression tester takes as input the EFGs of both the original and modified GUI, the valid initial states S_I for the modified GUI, and test cases for the original GUI. The checker partitions the original test suite into unusable and usable test cases. Importantly, it can also determine whether or not an unusable test can be repaired. Of the unusable test cases, the repaired test cases form a part of the regression test suite whereas the non-repairable ones are discarded. This new GUI regression testing method is summarized in Figure 2. New test cases, generated either to satisfy some coverage requirements for the test suite or to test those parts of the GUI that were not tested by the repaired test cases, are also a part of the regression test suite in addition to selected usable test cases.

Because software modifications from one version to another can be complex, it is impossible to develop an automated repairing technique for arbitrary modifications. This research achieves automation for a small class of GUI changes. We assume that events and windows: (1) have unique names (renaming can be carried out to accomplish this) and (2) are not renamed across versions of the GUI unless they are modified. For example, if an event `File` is not modified, then it is called `File` in the modified GUI. In case some events or windows are renamed, then the test designer is made aware of these changes by the GUI developer who must maintain a log of all such changes. Using these assumptions, we can automatically identify and classify GUI modifications as simple additions and deletions to the EFG. Similar approaches of tracking additions/deletions on control flow graphs for software have been used for incremental data-flow and code-optimizations [Pollock and Soffa 1992] and incremental testing [Harrold et al. 1992].

3.3.1 *Approach for Checking Test Cases.* The test-case checker's primary function is to identify unusable test cases and of those, which can be repaired. If the initial state S_0 of a test case is not one of the valid initial states S_I for the modified software, then it cannot be repaired. If $S_0 \in S_I$, then the test-case checker determines whether the event sequence in the test case is reusable by first identifying the modifications made to the GUI by comparing the EFGs of the original and modified GUIs.

If EFG_o and EFG_m are the EFGs of an original and modified GUI respectively, then the following sets of modifications are obtained by performing set subtraction. The functions $Vertices$ and $Edges$ return the sets **V** (the set of vertices) and **E** (the set of edges) for the EFG in question.

(1) The set of all vertices deleted from the original EFG:
vertices_deleted $\leftarrow Vertices(EFG_o) - Vertices(EFG_m)$;
(2) The set of edges deleted from the original EFG:
efg_edges_deleted $\leftarrow Edges(EFG_o) - Edges(EFG_m)$;

GUI modifications are recorded in two *bit vectors*, EDGES-MODIFIED and EVENTS-MODIFIED; each test case is associated with two bit vectors, EVENTS-USED and EDGES-USED. Determining whether a test case is usable/unusable is done by using very fast bitwise AND operations. Using this information, the test-case checker identifies test cases that were made unusable by each modification. For example, if an event e is deleted from the GUI, then all test cases that use event e are unusable. One GUI modification may be reflected in more that one set of modifications, and a test case may be marked as unusable several times because of the same modification. Being marked as unusable several times has no effect on the repairability of the test case. Once the unusable test cases have been identified, they are repaired by the test case repairer, which is described next.

3.3.2 *Approach for Repairing Test Cases.* The test-case repairing approach is based on user-defined transformations that delete or insert events into the test case at appropriate points. These transformations leverage the fact that an illegal event sequence uses at least one deleted event or edge. To develop the transformations that will make a GUI event sequence legal, we borrow an error-recovery technique from compiler technology; we skip events or try to insert a single new event until a legal event sequence is obtained [Aho et al. 1986]. This sequence can be found by skipping over events or by including events from the modified GUI.

If an event e_i, at position i in an event sequence is deleted from the GUI, then a transformation must remove e_i from the event sequence. However, to obtain a legal resulting event sequence, the transformation scans the event sequence from left to right, starting at position $i+1$, until it finds an event e_j such that either: (1) $< e_{i-1}; e_j >$ is a legal event sequence for the modified GUI, or (2) there is another event e_x, from the set of all the events in the modified GUI, such that $< e_{i-1}; e_x; e_j >$ is a legal event sequence for the modified GUI. Once such an e_j is found, then (for Transformation 1) the sub-sequence $< e_i; \ldots; e_{j-1} >$ is deleted from the event sequence and (for Transformation 2) e_x is inserted. Figure 3(a) shows these two transformations. In Transformation 1, the repairer searches for an event e_j from e_{i+1} to e_n, such that e_{i-1} `follows` e_j, and in Transformation

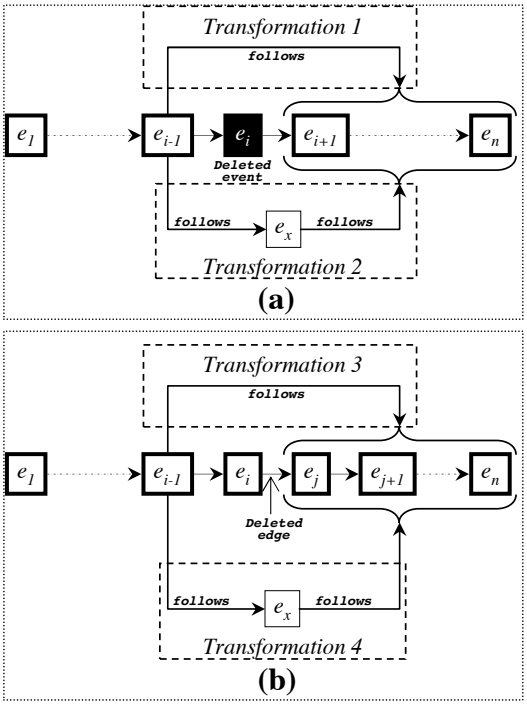


Fig. 3. Repairing an Event Sequence that Uses a (a) Deleted Event e_i, and (b) Deleted Edge (e_i, e_j).

2, it searches for an event e_x, from the set of all the events in the modified GUI, such that e_{i-1} `follows` e_x and for some e_j in the event sequence, e_j `follows` e_x. In general, these transformations may be extended to finding a sequence of events $< e_p; \ldots; e_q >$ such that $< e_{i-1}; e_p; \ldots; e_q; e_j >$ is a legal event sequence for the modified GUI.

Similarly, Figure 3(b) shows the transformations for the deleted edge (e_i, e_j). In these transformations, the event sequence is scanned from left to right, starting with the event e_j, the second element in the deleted edge. Transformation 3 tries to find an event e_a from the subsequence $< e_j; \ldots; e_n >$ such that e_a `follows` e_i. Transformation 4 tries to find an event e_x, from the set of all the events in the modified GUI, such that e_x `follows` e_i and e_j `follows` e_x.

Note that there may be more than one way to repair a test case; using all of them may yield several test cases, resulting in increased test-execution cost. A test designer may choose to limit the number of new test cases. In our implementation of the repairer, each event sequence is checked for all instances of deleted events and edges that made the event sequence illegal; when multiple ways are found, all of the repairs are used to produce more test cases. In principle, this approach may result in a large number of new test cases. In the worst case, if each of N transformations are applicable in M different places in a single test case, this approach would yield $M \times N$ new test cases.

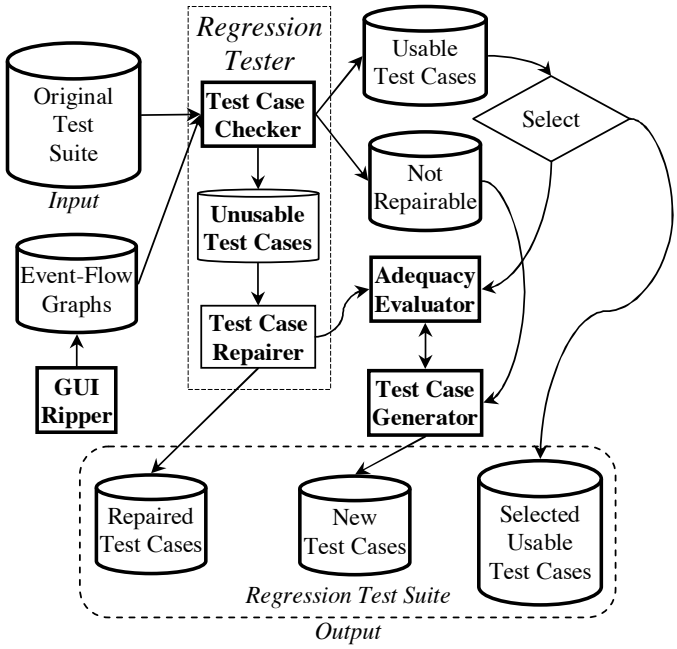


Fig. 4. Regression Tester's Components and their Interactions.

3.4 Tool Support

The regression tester has been implemented by us as a module in a test automation software called *GUI Testing frAmewoRk* (GUITAR). Other modules of GUITAR that are relevant to this study include a *GUI ripper* that automatically reverse engineers the GUI to create an EFG, a capture/replay tool to generate test cases, and an *adequacy evaluator* to evaluate the adequacy of test cases. Details of the design of the GUI ripper have been presented in earlier reported work [Memon et al. 2003]. Here, it is sufficient to understand that the GUI ripper employs a process called "GUI ripping" to automatically "traverse" the GUI by opening all its windows and extracting all the widgets, properties, and values. The output of the process is an EFG; it is checked and fixed manually, and used for regression testing. With the new regression testing module, the test designer first executes the regression tester to identify reusable test cases, and to repair the repairable test cases. The test designer then executes the adequacy evaluator to automatically determine those parts of the GUI that are not covered by the available (reusable and repaired) test cases, then uses the capture/replay tool to generate test cases to cover the missing parts.

Figure 4 shows the checker and repairer of the regression tester and their interactions. The figure also shows the interactions of these components with the test case generator and the adequacy evaluator to help generate new test cases that test parts of the GUI not tested by the available test cases [Memon et al. 2001; Memon et al. 2001]. Together, the repaired, new, and selected usable test cases form the

regression test suite.

4. EMPIRICAL STUDY

An earlier report of this research demonstrated, via a proof of concept study on two versions of a subject application (Adobe's Acrobat Reader versions 4.0 and 5.0), that the repairing approach is both practical and useful [Memon and Soffa 2003]. The study presented therein showed that of the 400 test cases generated for Version 4.0, 74% had become unusable for Version 5.0; 71.3% of these test cases were repaired in a matter of seconds.

This section extends the earlier work by presenting a detailed empirical study with the goal of determining whether GUI test cases for several evolving, multi-version, fielded GUI-based applications can benefit from the repairing technique. More specifically, the study is designed to empirically answer the following questions:

—**Q1:** How many test cases remain usable when an application changes? Of the unusable test cases, how many are repairable? How many do the four repairing transformations repair? Which transformations are more successful at repairing test cases?

—**Q2:** What are the characteristics of GUI test cases that make them unusable? Are certain types of test cases more prone to becoming unusable?

—**Q3:** What types of GUI changes are more likely to make large numbers of test cases unusable?

—**Q4:** Is a significant amount of test-development time saved by using the checker and repairer?

To answer the above questions, an empirical study is conducted using four popular open-source GUI-based applications available at SourceForge.net. Several versions of each application are downloaded. Test cases are generated for the first available version of each application. Given a test suite for version i (initially $i = 1$) of each application, the number of test cases that become unusable for version $i + 1$ is computed. The regression tester is deployed to obtain reusable and repaired test cases. Each version's test suite is enhanced by generating additional test cases to cover new functionality and existing functionality no longer covered by the unrepaired test cases, where coverage is evaluated in terms of GUI events. This process is repeated for all downloaded versions and findings reported to answer question **Q1**. Questions **Q2** and **Q3** are answered by characterizing unusable test cases and GUI changes, and examining those characteristics that contribute to test case unusability. The above study process, which utilizes both the automated checker and repairer, is named *Scenario 1*; Question **Q4** is addressed by creating three additional scenarios (Scenario 2 = {with checker, without repairer}, Scenario 3 = {without checker, with repairer}, Scenario 4 = {without checker, without repairer}) and determining the benefit, in terms of time, of using the checker and repairer to develop the regression test suite.

The following four applications were selected for this study:

1. **CrosswordSage**,[2] which is a tool for creating (and solving) professional

[2] `http://sourceforge.net/projects/crosswordsage`

looking crosswords with powerful word suggestion capabilities. When downloaded, it had an activity percentile of 98.21%. Versions 0.1, 0.2, 0.3.0, 0.3.1, 0.3.2, and 0.3.5 were downloaded for this study.

2. **FreeMind**,[3] which is a *mind-mapping*[4] software. It has an all time activity of 99.72%. Versions 0.0.2, 0.1.0, 0.4, 0.7.1, 0.8.0RC5 and 0.8.0 were downloaded.

3. **GanttProject**,[5] which is a project scheduling application featuring Gantt charts, resource management, calendaring, import/export (MS Project, HTML, PDF, spreadsheets). It has an all time activity of 98.12%. Versions 1.6, 1.9.11, 1.10.3, 1.11.1, and 2.0pre1 were downloaded.

4. **JMSN**,[6] which is a pure Java Microsoft MSN Messenger clone, featuring instant messaging, file send/receive, MSNlib (for developers), and chat logging. It has an all time activity of 98.93%. Versions 0.9a, 0.9.2, 0.9.5, 0.9.7, 0.9.8b7, and 0.9.9b1 were downloaded.

These applications were selected because their code and GUI characteristics, and development history made them interesting subjects for this study. In particular, CrosswordSage was selected due to the simple design of its first version's GUI (Version 0.1 has 1 window) and rapid code-churn rate; in just three months, it has evolved to a 7-window GUI application with `Help`, `Instructions`, and `About` menus, and associated windows. Due to these rapid changes, developers have added as well as removed GUI features across versions. This style of evolution enables the study of the effect of quick changes (particularly widget deletions that are less common in other applications) on GUI regression testing. FreeMind was selected due to its complexity – its latest version has 32 windows and 1095 widgets. Regression testing of this application will enable the study of the performance and scalability of the regression testing techniques and algorithms.

GanttProject was selected since it has evolved steadily over a long period of time (3 years on Sourceforge.net). The developers have been careful not to make many changes to the GUI quickly. Version 1.6 was the first public release with English as the default language. Version 1.9.11 focused on fixing bugs and improving usability. More specifically, widgets were added for customization of fonts, the language menu was removed, and windows were added to support export, opening, and saving of files from/to the server. This version has 10 windows; the latest version (released after almost 2 years) has only four new windows. However, the overall code-base has more than doubled (to approximately 64 KLOC) in that period. These characteristics of the application will enable the study of the effect of carefully planned GUI modifications on regression testing. JMSN was selected due to its relatively constant number of windows (5 in all versions except the latest, which has 6), which will enable the study of the effect of intra-window modifications on regression testing. Finally, only Java Swing applications were selected because the tools in GUITAR such as the GUI ripper work well with applications whose GUIs have been implemented using Java Swing.

The detailed characteristics of the applications/versions are shown in Table II.

[3] `http://sourceforge.net/projects/freemind`
[4] `http://en.wikipedia.org/wiki/Mind_map`
[5] `http://sourceforge.net/projects/ganttproject`
[6] `http://sourceforge.net/projects/jmsn`

Subject Applications & Versions	Code Attributes				GUI		EFG		EFG Changes			
									Added		Deleted	
	Files	LOC	Classes	Methods	Windows	Widgets	Vertices	Edges	Vertices	Edges	Vertices	Edges
CrosswordSage												
0.1	2	369	4	10	1	12	10	91	4	45	2	14
0.2	2	445	4	13	3	15	12	122	47	675	4	34
0.3.0	11	2025	20	124	4	73	55	763	1	5	1	5
0.3.1	11	2084	20	129	4	73	55	763	8	215	9	156
0.3.2	11	2248	22	144	3	68	54	822	35	807	4	190
0.3.5	19	3617	36	244	10	117	85	1439				
Freemind												
0.0.2	28	4445	76	302	4	110	82	1468	13	616	2	213
0.1.0	41	5801	105	394	4	118	93	1871	95	2383	1	323
0.4	66	13453	175	800	6	234	187	3931	221	9961	3	639
0.7.1	92	21983	258	1415	16	501	405	13253	400	38373	2	1557
0.8.0RC5	509	101981	1503	5285	32	1094	803	50069	5	2250	1	910
0.8.0	509	102225	1504	5295	32	1095	807	51409				
GanttProject												
1.6	16	4542	25	381	5	126	87	1153	95	2701	0	331
1.9.11	159	24143	269	1280	10	238	182	3523	21	568	1	415
1.10.3	233	31947	380	1751	12	268	202	3676	5	334	1	201
1.11.1	318	39138	526	2384	13	282	206	3809	19	941	0	561
2.0.pre1	454	64312	771	3852	14	311	225	4189				
JMSN												
0.9a	32	6411	46	332	5	69	54	961	6	278	2	190
0.9.2	33	6749	47	353	5	73	58	1049	3	199	1	152
0.9.5	45	9485	60	487	5	75	60	1096	4	378	1	319
0.9.7	46	9774	61	495	5	79	63	1155	13	579	1	218
0.9.8b7	49	10714	65	533	5	95	75	1516	7	488	0	309
0.9.9b2	50	11290	68	556	6	122	82	1695				

Table II. Characteristics of Subject Applications

The columns grouped under `Code Attributes` in the table show that these applications have non-trivial sized code-bases. Column `Files` shows the number of Java source code files, `LOC` shows the number of (non-comment) source lines, and `Classes` and `Methods` show the number of classes and methods used to implement the applications respectively. The numbers show that all four applications provide a good mix of small, moderate, and large application code sizes. The columns grouped under `GUI` show that all the applications have multiple windows with many widgets.

4.1 Study Procedure

The EFGs for all versions of the applications were obtained using the GUI ripper; test cases (satisfying the *event* and *event-interaction* adequacy criteria [Memon et al. 2001]) were obtained for the first version of each application, and, for each subsequent version $i > 1$ of each application, the following steps were repeated:

—*Step 1:* Compute the number of usable test cases from the test suite of version $i - 1$.

—*Step 2:* Of the unusable test cases, determine the number of repairable test cases and repair them.

—*Step 3:* Generate additional test cases for version i to satisfy the event and event-interaction adequacy criteria.

Details of the above steps are presented in subsequent paragraphs. Addressing question **Q4** involves the execution of additional steps, which will be presented in Section 4.6.

Obtaining EFGs and Computing Changes: The GUI ripper was executed on all versions of the applications to obtain EFGs. The columns grouped under `EFG`

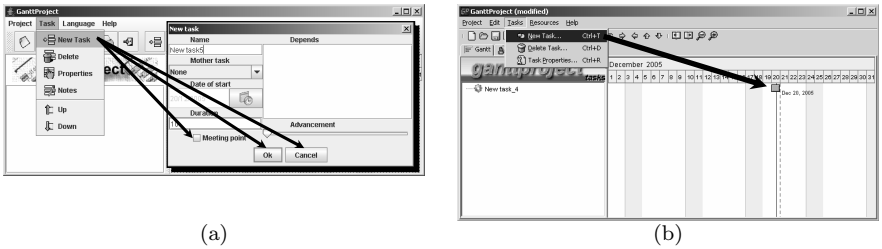
Subject Applications & Versions	TCG First Version	GUI Ripper		Event Matching		Checker	Computing Unusable Test-cases w/o Checker	Repairer	Add-TCG	Supplemental Test cases	Total Time			
	Manual	Auto	Manual	Auto	Manual	Auto	Auto	Auto	Manual	Manual	Scenarios			
	hrs.	sec.	min.	sec.	min.	ms.	min.	sec.	hrs.	hrs.	1	2	3	4
CrosswordSage														
0.1	2.01	0.2	0.25											
0.2		0.5	1.1	0.24	4	0	16.34	0.04	0.73	6.21	0.81	7.02	1.08	7.21
0.3.0		0.6	1.4	0.18	11	1	19.41	0.04	16.44	30.63	16.65	47.28	16.97	47.39
0.3.1		0.6	1.4	0.18	18	1	129.48	0.31	6.34	38.03	6.67	44.70	8.83	46.53
0.3.2		0.4	1.6	0.18	18	1	119.3	0.27	5.05	32.97	5.38	38.35	7.37	40.01
0.3.5		1.9	3.5	0.28	23	1	118.18	0.34						
Freemind														
0.0.2	56.01	0.8	2.1											
0.1.0		0.9	1.8	0.31	29	2	415.75	0.59	29.96	52.63	30.48	83.11	37.41	89.52
0.4		1.5	2.1	0.62	47	4	537.41	0.51	104.93	182.69	105.74	288.43	114.70	296.58
0.7.1		4.2	4.5	1.35	99	13	856.4	1.14						
0.8.0RC5		8.1	10.1	2.68	201	50	3165.11	4.01						
0.8.0		9.1	12	2.69	268	51	12244.8	11.19						
GanttProject														
1.6	26.36	0.8	1.8											
1.9.11		2.1	3.5	0.61	45	4	233.49	0.28	136.92	169.38	137.73	307.11	141.62	310.19
1.10.3		3.2	4.2	0.67	64	4	874.02	0.99	41.86	105.94	43.00	148.94	57.57	162.37
1.11.1		2.5	4.5	0.69	68	4	945.17	1.07	21.1	119.49	22.31	141.80	38.06	156.34
2.0.pre1		3.2	5.1	0.75	72	4	863.03	0.99						
JMSN														
0.9a	15.47	1	2.4											
0.9.2		0.9	2.2	0.19	19	1	194.23	0.18	8.91	19.79	9.26	29.05	12.50	31.94
0.9.5		0.7	2.1	0.20	20	1	282.77	0.22	3.69	15.79	4.05	19.84	8.77	24.19
0.9.7		0.9	2.5	0.21	21	1	355.6	0.20	3.14	23.47	3.52	26.99	9.45	32.54
0.9.8b7		0.8	1.8	0.25	23	2	315.6	0.25	14.06	25.74	14.47	40.21	19.73	45.06
0.9.9b2		1.1	3.1	0.27	26	2	594.44	0.32						
Scenario 1		x	x	x	x	x		x	x		1			
Scenario 2		x	x	x	x	x			x	x		2		
Scenario 3		x	x	x	x		x	x	x				3	
Scenario 4							x		x	x				4

Table III. Summary of Computation Times.

of Table II show the characteristics of the EFGs. The number of vertices in the EFG is strictly less than the number of widgets in the GUI because EFG vertices represent only those widgets on which user events can be executed (*e.g.*, text-labels that have no associated event handlers are excluded). The results of this process were verified manually. The time taken for the overall ripping and verification is shown in column `GUI Ripper` of Table III. Once all the EFGs were available, the GUI ripper used a number of heuristics to "match" the events of one version to the next. As discussed in Section 3, the heuristics are based on the assumption that widget names remain unchanged unless they have been modified. The GUI ripper first matched windows between two subsequent versions; this matching was verified manually; if it was found to be incorrect then it was fixed. The events within each window were matched next. This matching was also verified and fixed manually. Note that this matching step and manual work may be avoided by maintaining a list of GUI changes during development.

The total time required for matching and verification is shown in column `Event Matching` of Table III. Once the matching had been finalized, the sets of added/deleted vertices and edges were computed. The sizes of these sets is shown in columns grouped under `EFG Changes` in Table II. EFG changes made to a version are shown in the row corresponding to that version.

Table II shows that GUI modifications lead to a large number of edge deletions. Deletions of vertices are less common. The rapid changes to CrosswordSage caused relatively more deletions of vertices than in other applications. Even if features are not eliminated across versions, layout changes still cause a non-trivial number of edge deletions. For example, even though zero vertices were deleted in Version 1.6 of GanttProject to obtain Version 1.9.11, 331 edges were deleted due to lay-



(a) (b)

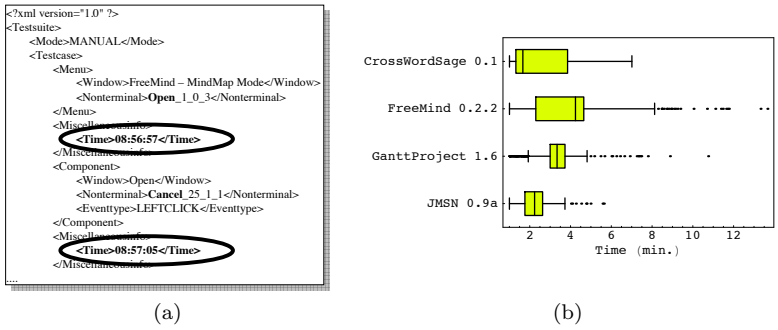
Fig. 5. An Example Layout Modification in GanttProject (a) Version 1.6 to (b) Version 1.9.11

out changes. One such modification is shown in Figure 5; the event `New Task` no longer opens the `New task` window (Figure 5(a)); instead, it creates a new node representing a task (Figure 5(b)). All edges from `New Task` to each event (corresponding to each widget) in the `New task` window are deleted. Some examples of these edges shown in Figure 5(a) include (`New Task`, `Ok`), (`New Task`, `Cancel`), and (`New Task`, `Meeting point`), where events `Ok`, `Cancel`, and `Meeting point` correspond to widgets in the `New task` window.

Test-Case Generation: Test cases were generated for the first version of each application using a capture/replay tool. Four testers were employed for this process – each was assigned to one application. The testers were asked to spend 1-4 hours to familiarize themselves with the assigned subject application. The test-case generation process was carried out in several steps. Testers were first assigned several *tasks*, *i.e.*, activities that they could complete using the application. The tasks were chosen carefully so that they covered most of the functionality of the applications. In all, 25, 400, 200, and 150 tasks were assigned for CrosswordSage, FreeMind, GanttProject, and JMSN, respectively. The testers generated a sequence of events to complete each task; the capture tool recorded the user interaction as a test case. An *initial test set* was obtained in this way. The adequacy analyzer based on event and event-interaction adequacy criteria developed in earlier reported work [Memon et al. 2001] was then executed, which generated a report summarizing the parts of the GUI missed by the test cases. The report was in the form of edges of the EFG that were not covered by the initial test set. The testers were then asked to generate additional test cases to cover these edges without using tasks. They took 1, 9, 5, 4 day(s) on CrosswordSage, FreeMind, GanttProject, and JSMN respectively to complete the entire process.

There are several points to note about the test case generation process. First, to record each test case, the tester launched the application from scratch. Each test case started with an event in the main window and ended with the `Exit` event. Second, the process described earlier ensures that the test suite satisfies the event and event-interaction adequacy criteria; these criteria require that each vertex and edge in an EFG is covered by at least one test case in the suite. Satisfying them guarantees that the effect of a change to any vertex/edge is observed during regression testing. Note that these criteria provide an objective metric to quantify

the "rules of thumb" already advocated by GUI testing experts,[7] *e.g.*, "*For Each Window in the Application* [perform some action on widgets]" and "*On every Screen* [perform some validation steps]." Our own earlier findings also support such rules of thumb and, in fact, show that source-code coverage may be misleading for GUI testing [Memon et al. 2001]. 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 (reached by executing different permutations of events) of the software each piece of code is tested. Third, as discussed in Section 1, this work assumes the existence of "global test oracles" common to all test cases, rather than ones associated with each individual test case; hence, no test oracle was generated for these test cases. Finally, the test suites for CrosswordSage, FreeMind, GanttProject, JMSN contained 51, 879, 497, and 418 test cases respectively.



(a) (b)

Fig. 6. (a) Time-stamps in a Test Case and (b) Time Taken to Record Each Test Case.

The test cases recorded by GUITAR contain time-stamps for each event. Total time is the sum of all the times spent per event in the test case. Figure 6(a) shows a part of a test case generated for FreeMind; two events `Open` in the `File` menu, followed by `Cancel` in the `FileOpen` window are shown. During capture, the clock-time (enclosed within the <`Time`> tag) when each event was executed is also stored in the test case. These time-stamps were used to obtain the total time required to generate the test case. Figure 6 shows the time it took to generate the test cases as distributions. The box-plots provide a concise display of each distribution. The line 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 90% of the distribution; outliers are shown as points beyond the whiskers. The plots show that test-case generation time varied between applications; most test cases were generated in 2-4 minutes. The total time spent generating test cases is shown in column `TCG First Version` in Table III.

Step 1: Executing the Checker. The checker took as input the test suites for version i (initially $i = 1$) of each application, and the set of modifications

[7] `http://members.tripod.com/bazman` and `http://www.rspa.com/checklists/guitest.html`.

(additions/deletions of EFG vertices/edges) made to it to obtain version $i+1$. The checker then computed the number of usable, unusable, and repairable test cases. The total time taken by the checker is shown in column `Checker` in Table III.

Step 2: Executing the Repairer. Each repairable test case was then repaired using the four transformations. The total time taken by the repairer is shown in column `Repairer` in Table III.

Step 3: Generating Additional Test Cases. Because several test cases could not be repaired, and each version added new features, the test suite for each version needed to be updated to satisfy the test adequacy criteria. New tasks were developed to test the newly added functionality. The testers were then asked to generate new test cases to complete the tasks. The adequacy evaluator was then executed to determine the adequacy of the repaired, reusable, and new test cases. As before, the output of the adequacy evaluator was a set of edges of the EFG not covered by the test cases. The testers then generated additional test cases to cover all the missing edges. The total time taken to generate the additional test cases is shown in column `Add-TCG` in Table III; the time distributions are summarized in Figure 7. The new, repaired, and reusable test cases for version i were used as the regression test suite for version $i+1$.

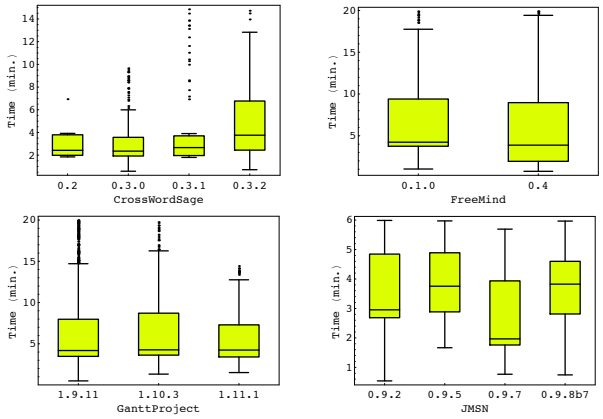


Fig. 7. Time Taken to Record Each Additional Test Case.

The only exception to the above test case generation process were Versions 0.7.1 and 0.8.0RC5 of FreeMind. The complexity of these versions require a prohibitively large number of test cases. Instead of using a capture/replay tool to perform this task, an automated model-based test case generator (also a part of GUITAR) was used to obtain the test cases. Details of this test-case generator have been presented in earlier reported work [Memon et al. 2005; Memon and Xie 2005]. The tool uses graph traversal techniques to cover edges of the EFG.

4.2 Threats to Validity

As with any empirical study, this study raises several threats to validity. Results should be interpreted with these threats in mind.

Threats to internal validity are problems with the study design that may alter the cause-effect relationship being studied. The biggest threats to internal validity are related to the way we create test cases and evaluate their coverage. We used one technique to generate test cases – capture/replay tools; in one case we used EFGs with an automated test-case generator. We used event and event-interaction coverage as a measure of adequacy. Other techniques (*e.g.*, using programming the test cases manually) and/or using different test criteria (*e.g.*, code coverage) may produce different types of test cases, yielding different results.

Threats to construct validity arise whenever some measurement is used as a proxy for the real value of interest. In this study, our measure of cost (in terms of wall time) combines human effort and execution cost. A more accurate measure would use domain-specific knowledge to assign appropriate "weights" to these cost components.

Threats to external validity [Wohlin et al. 2000] are conditions that limit the ability to generalize the results of our experiment to industrial practice. Our subject applications, the experimenters used to generate test cases, and choice of GUI-tasks are the biggest threats to external validity. First, we have used four applications, downloaded from SourceForge as our subject applications. Although they have different types of GUIs, this does not reflect the wide spectrum of possible GUIs that are available today. Second, all our subject programs were developed in Java. Although our abstraction of the GUI maintains uniformity between Java and non-Java applications, the results may vary for non-Java applications. Third, our GUIs are static, in that we do not have widgets that are created on-the-fly from back-end data. We expect that our repairing algorithms need to be improved for such "dynamic" GUIs. Finally, the timing for the manual parts of the approach (e.g., checking, fixing, test-case generation) is likely to be highly dependent on the specific tester considered; given the small number of data points in the study, these numbers could vary considerably.

4.3 Addressing **Q1**: Studying Test Case Unusability, Repairability, and Effectiveness of the Four Transformations

The results (also the answers to question **Q1**) of the three steps of Section 4.1 are summarized in Table IV. The columns grouped under `Checker` show the results of Step 1, `Repairer` show the results of Step 2, and `New Generated` show the results of Step 3. As the table shows, for most application versions (with the exception of JMSN), more than half (in one case 86.3%) of the test cases became unusable. This result was interesting as most of these subject applications did not remove a large number of widgets across versions; only the deleted edges in the EFGs caused a large number of test cases to become unusable. Due to unplanned and rapid GUI design evolution of CrosswordSage, the percentage of unusable test cases (69.3%–86.3%) was larger than for other applications. On the other hand, planned changes and relatively fewer widget deletions in the GUIs of FreeMind and GanttProject resulted in a smaller percentage of unusable test cases – 46.9%–66.6%. In case of JMSN, more than half the test cases remained reusable. There was no noticeable difference between the results obtained for the test cases generated using the automated test-case generator (for Versions 0.7.0 and 0.8.0RC5 of FreeMind) and the capture/replay tool.

Subject Applications & Versions	Existing Test Cases	Checker					Repairer										Regression Suite		Generated Manually Scenario 2
		Reusable		Unusable		Repairable	Repaired		Not Repaired		Number Obtained	Percentage Increase	Transformations Applied				Repaired & Reusable	New Generated	
		#	%	#	%	%	#	%	#	%			T1	T2	T3	T4			
CrosswordSage																			
0.1																		51	
0.2	51	13	26	38	74	100	25	67	13	34	30	122	50.0	6.7	33.3	10.0	43	15	67
0.3.0	58	18	31	40	69	100	21	54	19	48	23	110	52.2	0.0	30.4	17.4	41	323	212
0.3.1	363	50	14	314	86	100	216	69	98	31	226	105	43.8	5.8	43.8	6.6	276	88	312
0.3.2	363	97	27	267	73	100	178	67	89	33	238	134	45.0	5.9	43.7	5.5	335	57	259
0.3.5	391	54	14	337	86	100	171	51	166	49	241	141	37.8	3.7	37.8	20.7	295		
Freemind																			
0.0.2																		879	
0.1.0	879	293	33	586	67	100	316	54	270	46	328	104	51.8	1.8	43.0	3.4	621	270	498
0.4	891	385	43	506	57	100	389	77	117	23	427	110	52.9	5.9	28.8	12.4	812	1060	1212
0.7.1	1872	730	39	1142	61	100	582	51	560	49	698	120	36.0	1.9	34.0	28.2	1428	4883	
0.8.0RC5	6311	2298	36	4013	64	100	2006	50	2007	50	2246	112	49.0	5.0	26.0	20.1	4544	19298	
0.8.0	23842	12653	53	11189	47	100	7944	71	3245	29	10247	129	52.2	10.0	35.0	2.8	22900		
GanttProject																			
1.6																		497	
1.9.11	497	214	43	283	57	100	155	55	128	45	181	117	43.6	5.5	26.0	24.9	395	1283	1609
1.10.3	1678	692	41	986	59	100	502	51	484	49	677	135	40.9	8.0	26.9	24.2	1369	382	1160
1.11.1	1750	680	39	1071	61	100	717	67	354	33	910	127	57.9	10.0	32.0	0.1	1590	224	1334
2.0.pre1	1814	828	46	986	54	100	502	51	484	49	597	119	48.9	4.9	33.8	12.4	1425		
JMSN																			
0.9a																		418	
0.9.2	418	240	57	178	43	100	96	54	82	46	115	120	53.9	7.8	33.9	4.3	355	145	330
0.9.5	499	282	57	217	43	100	110	51	107	49	183	167	59.6	6.6	24.6	9.3	465	57	291
0.9.7	522	319	61	203	39	100	135	67	68	33	163	121	54.6	2.5	39.9	3.1	482	68	370
0.9.8b7	550	301	55	249	45	100	131	53	118	47	187	143	41.7	3.7	34.8	19.8	488	234	390
0.9.9b2	722	397	55	325	45	100	191	59	134	41	250	131	48.8	2.8	36.0	12.4	647		

Table IV. Summary of Regression Tester's Results

Because the initial state in all the test cases remained valid, all the test cases were repairable. Of the unusable test cases, a large percentage (more than 50% in most cases; in one case 77%) were repaired in a few seconds using the four simple transformations. Because a test case may be repaired in multiple ways, an unusable test case may yield multiple test cases. In one case, there was a 167% increase in the size of the repaired suite. Not all four transformations were equally effective at repairing test cases; the column `Transformations Applied` under `Repairer` shows the percentage of transformations that yielded the repaired test cases; this column shows that T1 and T3 were quite effective, followed by T4, and finally T2. For improved visualization, the data is plotted in Figure 8 with the transformations ordered, on the x-axis, as T1, T3, T4, and T2; the y-axis shows the percentage of cases in which a transformation was applied.

An analysis of variance (ANOVA) test (with Alpha = 0.05) was conducted to determine whether the differences in applicability between the transformations are statistically significant. The null hypothesis was that the means of all transformation data sets are equal; the alternative hypothesis was that the means of the data sets are different (not equal). The observed p-value was much less than 0.05 (it was in fact 6.56×10^{-34}); hence, the null hypothesis was rejected and the alternative hypothesis was accepted. Given that the ANOVA test had determined that the means were statistically different, the Tukey *post hoc* means comparison test was subsequently used to compare all possible pairs of means; this test showed that the differences between each transformation pair was also statistically significant.

Examining the unrepaired test cases demonstrated that there is considerable room for improvement in terms of the set of repairing transformations. The four currently used transformations completely ignore the hierarchical structure of the GUI, except that implicitly encoded in the EFGs. New transformations that consider this hierarchy and insert/delete event sub-sequences using "sub-trees" of the GUI's hierarchical structure would be particularly useful. For example, if two windows `Find` (invoked using `find`; terminated using `OKFind`) and `Replace` (invoked using `replace`; terminated using `OKReplace`) are merged into one window

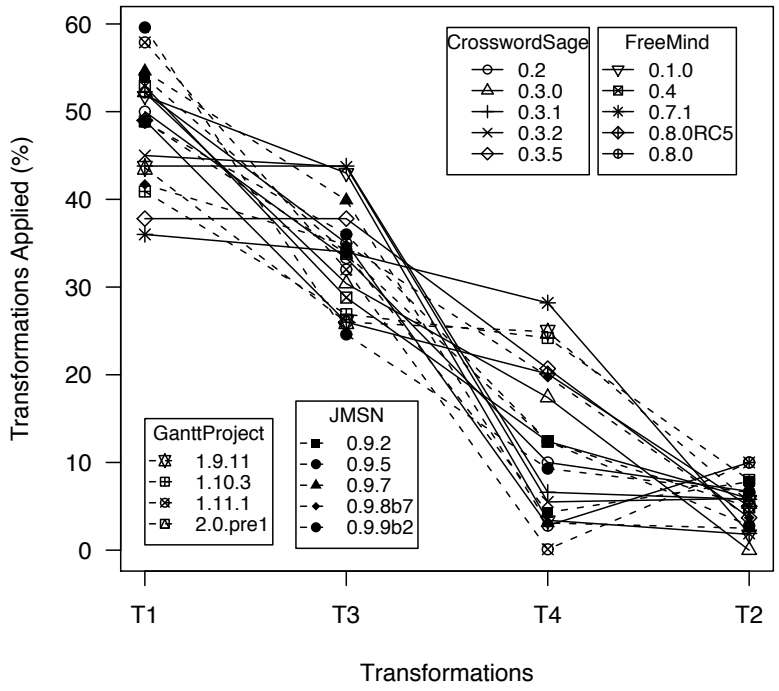


Fig. 8. Transformation Type and Percentage Applied on Each Application Version.

`Find/Replace` (invoked using `find/replace`; terminated using `OKfindreplace`), then a test case that contains a subsequence starting with the `find` event and ending with the `OKfindreplace` may be repaired by viewing the `Find` window as a "tree" rooted at `find`, and replacing the sub-sequence by the new appropriate `find/replace` invoking and `OKfindreplace` termination events.

In summary, to answer question **Q1**, these results showed that the unusability of GUI test cases is a serious problem; simple layout changes (not only removal of features) is sufficient to make large numbers of test cases unusable. The regression tester helped to identify unusable test cases within seconds and make them usable. Even for the large GUI size of FreeMind, the entire process took a few seconds per test case.

4.4 Addressing **Q2**: Studying the Impact of Test Case Characteristics on Its Unusability

To answer question **Q2**, an informal examination of test case characteristics and number of unusable test cases revealed that short test cases (*i.e.*, those with fewer events) were more likely to remain usable across versions. This section summarizes the detailed analyses that ensued.

To study the impact of a test case's length on its unusability, all the test suites were partitioned into equivalence classes. A test case x is in equivalence class i if and only if $Length(x) = i$, where $Length()$ is a function that returns the number of events in the test case. The result of this partitioning is shown as a set of column

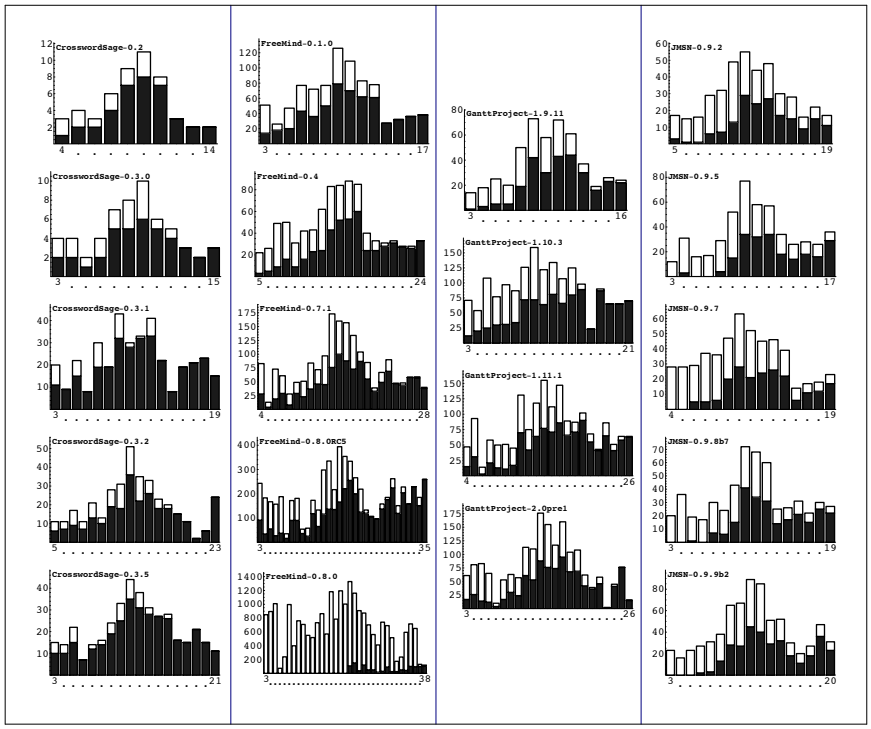


Fig. 9. Affect of Length on Test Case Unusability.

graphs in Figure 9. The x-axes show the equivalence class, represented by an integer indicating the length of its constituent test cases. The y-axes show the sizes of the equivalence classes; the total height of each column shows the total number of test cases in the equivalence class. Each equivalence class is further partitioned into unusable (shaded part of each column) and usable (unshaded part of the column) test cases. For added clarity, the same data is presented in Figure 10 by computing the percentage of usable test cases per column (unshaded part).

The data was further prepared to determine whether the effect of test-case length on its unusability was statistically significant. In particular, test cases per application version were partitioned into five buckets (A, B, C, D, and E). Bucket A contained the shortest 20% of the test cases, Bucket B contained the next 20% by length, and so on. Once all buckets for an application version had been filled, the "average percentage usable" metric was computed for each bucket. This process was repeated for each application version. The result is summarized in Figure 11. This figure shows that long test cases are more likely to become unusable across versions; a large percentage of short test cases remain usable. This result is directly related to the fact that because longer test cases cover a larger number of EFG edges than short test cases, the probability that they contain an edge that has been deleted is higher.

As was done earlier, an ANOVA test (with Alpha = 0.05) was conducted to

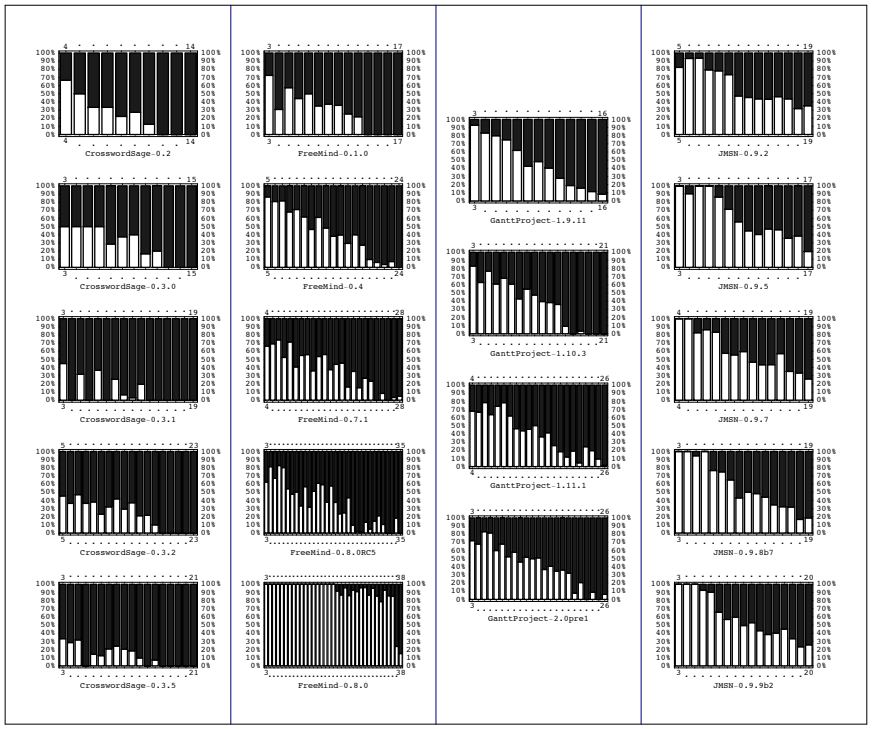


Fig. 10. Percentage of Test Cases that Remained Usable (unshaded part).

determine whether the differences between the test case length partitions are statistically significant. The null hypothesis was that the means of all data sets are equal; the alternative hypothesis was that the means of the data sets are not equal. The observed p-value was 5.51×10^{-13}, *i.e.*, much less than 0.05; hence, the null hypothesis was rejected and the alternative hypothesis was accepted. A subsequent Tukey *post hoc* means comparison test showed that the differences between each pair of partitions was also statistically significant. This analysis and the observed data plots showed that long GUI test cases are more likely to become unusable, answering question **Q2**,

4.5 Addressing **Q3**: Studying the Impact of GUI Changes on Test Case Unusabililty

The results shown thus far have demonstrated that widget removal and/or restructuring (*i.e.*, vertex and edge deletions in the EFG) cause a large number of test cases to become unusable. An informal examination of the data showed that modifications to certain widgets impacted a large number of test cases than others. A more formal treatment of this observation is summarized in this section.

To study the number of test cases that were impacted by specific edge deletions (note that edge deletions capture both widget removal and restructuring), the number of unusable test cases were plotted against deleted edges in Figure 12. The x-axis of each plot represents deleted edges, each represented by a unique num-

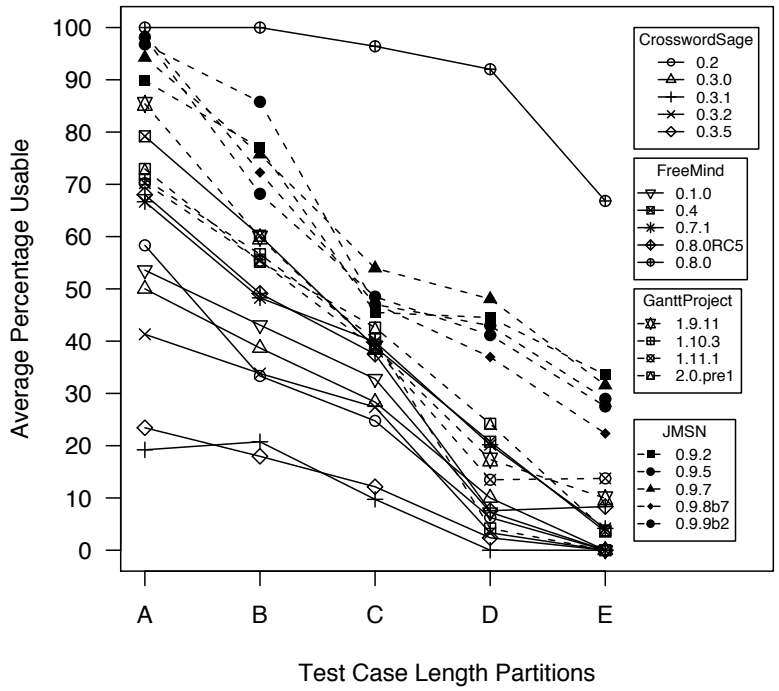


Fig. 11. Average Percentage of Usable Test Cases Partitioned by Length.

ber. The y-axis represents the number of impacted test cases. A point (x, y) on this plot shows that y number of test cases were rendered unusable because of the deletion of edge x. Note that because a test case may become unusable due to the deletion of several edges, it is counted several times. For improved presentation, the points are sorted in order of increasing y-axis values, and the $y = 1$ values are not plotted; their number is summarized as a label "`Singles =`" on the plot.

All the plots in Figure 12 share a number of properties. First, a large number of points lie close to the x-axis; indeed, the majority of them are singles, *i.e.*, number of impacted test cases is 1; the number decreases significantly with increasing distance from the x-axis. This result shows that most edge deletions affect very few test cases. Second, the sorting of the points in order of increasing y-axis values gives the visual impression that the points lie in clusters in near horizontal "lines" with significant gaps between adjacent lines. This result shows that some edge deletions impact a very large number of test cases. For example, a single edge deletion rendered more than 400 of the 508 test cases unusable in FreeMind-0.4. Similarly, a vast majority of the 283 unusable test cases for GanttProject-1.9.11 became unusable due to just four edge deletions.

The results of Figure 12 are best explained by examining the hierarchical structure of GUIs. More specifically, GUI widgets are arranged in a hierarchy of menus and windows. One such hierarchy is shown in Figure 13. The bitmap is an annotated part of a screen-shot of the GUI ripper's output. Figure 13 shows some

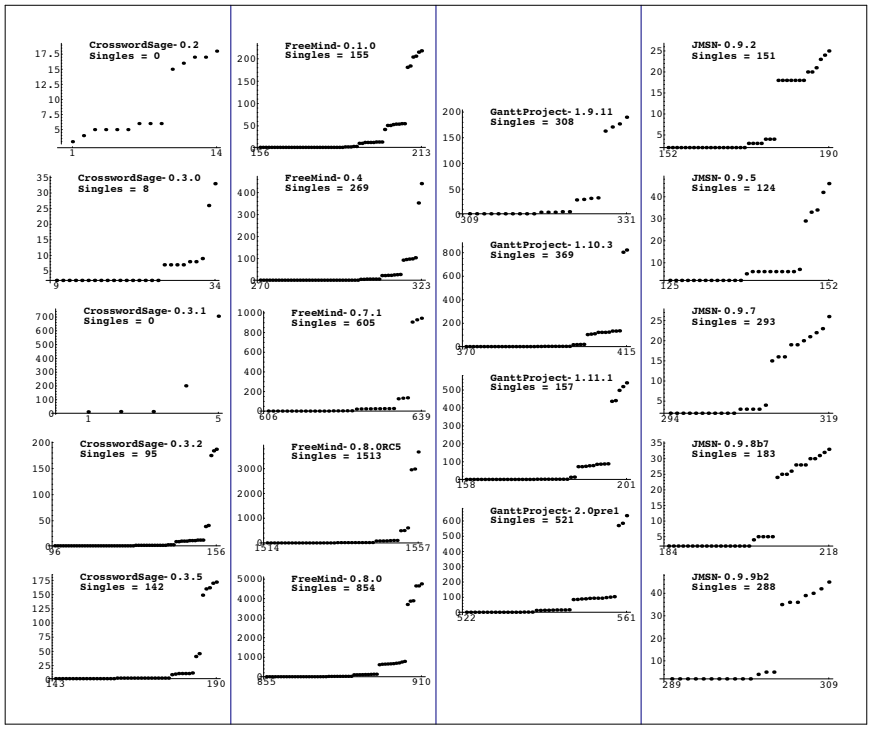


Fig. 12. Impact of Deleted EFG Edges on the Number of Unusable Test Cases.

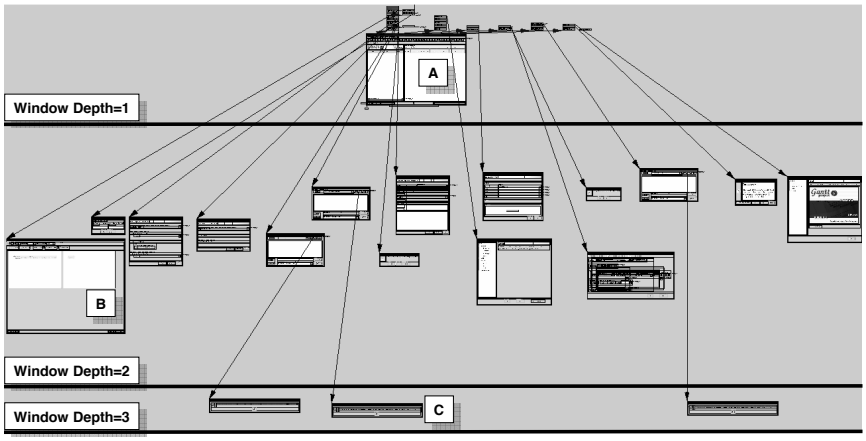


Fig. 13. An Example GUI Windows Hierarchy.

of the windows and their hierarchical relationship of GanttProject-2.0pre1. The main window (labeled `A`) is first available to a user when the software is launched; it is said to be at *window depth of 1.* All other windows are available directly or indirectly via the main window. Two other windows have been labeled as `B` and `C`

respectively. Window `B` is at depth 2 and `C` is at depth 3.

All testers must interact with window `A`. Hence each test case must contain events from this window. Changes made to this window have the potential to impact a large number of test cases. As testers perform different tasks, they use events from other windows; however, not all test cases will contain events from all windows. For example, few test cases will use the events of window `B`. A change made to the events on window `B` is less likely to affect many test cases. Finally, changes made to window `C` are least likely to affect many test cases as very few test cases will contain these events. Going back to Figure 12, the cluster of points with largest y values correspond to edges that are contained in the depth 1 window, usually in the pull-down menu. The second cluster corresponds to changes made at the depth 2 windows and second level menus, and so on. The above observations also explain the relatively smaller percentage of unusable test cases for JMSN (less than 50%). Very few events were modified in the main window of JMSN, causing a smaller percentage of test cases to be affected.

The observation that the *location* of a modified event in the GUI hierarchy can cause different numbers of test cases to become unusable may be used to quantify GUI changes in terms of their potential to impact test cases. An event (such as `Edit` in the pull-down menu) that must be executed by a user to reach other events (such as `Cut`) may be called a *dominator event* (the term "dominator" has been borrowed from compiler theory [Lengauer and Tarjan 1979]). Most pull-down menu events that are used to open sub-menus and child windows are common examples of dominator events. Each dominator event may be assigned a "dominator weight" indicating the number of events that it "dominates." Similarly, some events behave as "post-dominators," *i.e.*, users must execute these "termination" events to exit from certain dialogs and the application. Common examples include `Ok`, `Cancel`, and `Exit` events.

To further study this concept, each deleted edge shown in Figure 12 was assigned its dominator weight. The weight of each edge (x, y) was computed as the larger of the number of events that x and y dominate. The `Cancel` termination event was ignored in this analysis; for each `OK` type of termination event (each modal dialog in our subject applications has two termination events – `Cancel` and `OK`), its dominator weight was computed as the number of events that require `OK` to be executed *after* their execution.

The dominator weight of each deleted edge was then plotted against the number of test cases that become unusable for that edge. The results are summarized in Figure 14; the x-axes show weights, the y-axes show the number of impacted test cases. The points in these plots show that edges that have a large weight impact a large number of test cases. The line in each plot is a linear least-squares fit to the data. The R-square value (shown with each plot) is an indicator of how well the linear model fits the data (*e.g.*, an R-square value close to 1.0 indicates that we have accounted for almost all of the variability with the variables specified in the model). Figure 14 shows that most R-square values are well above 0.90.

In order to study whether the effect of event dominator weight on a test case's unusability was statistically significant, test cases (per application version) were partitioned into two buckets (W1 and W2). Bucket W1 contained the average

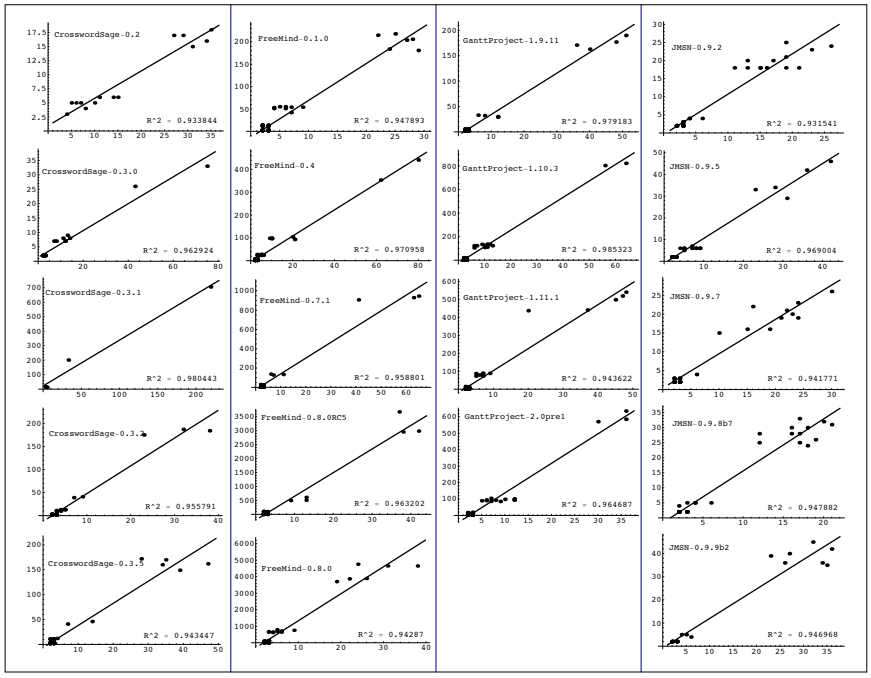


Fig. 14. "Dominator Weight" of Edges Deleted vs. Number of Unusable Test Cases

number of test cases that were made unusable by events of dominator weight w_1, where $(MinDom \leq w_1 \leq (MinDom + \lceil \frac{MaxDom - MinDom}{2} \rceil))$; Bucket W2 contained the average number of test cases that were made unusable by events of dominator weight w_2 where $(MinDom + \lceil \frac{MaxDom - MinDom}{2} \rceil) < w_2 \leq MaxDom)$; $MinDom$ is the smallest dominator weight and $MaxDom$ is the largest. Intuitively W1 contained all "relatively small dominator weight" test cases and W2 contained the "relatively large dominator weight" cases. This process was repeated for each application version. The result is summarized in Figure 15. The plot shows that GUI changes made to dominator and post-dominator events with large weights are more likely to cause a large number of test cases to become unusable.

Because there are only two data sets, a paired-t test (with Alpha = 0.05) was conducted to determine whether the differences between the event dominator partitions are statistically significant (the pairing in the test is because both data sets were obtained from the same software subjects). The null hypothesis was that the means of both data sets are equal; the alternative hypothesis was that the means are not equal. The observed p-value was 0.012, *i.e.*, less than 0.05; hence, the null hypothesis was rejected and the alternative hypothesis was accepted. This analysis, together with the observed data plots, leads to the conclusion that changes to events with large dominator weights are more likely to cause a large number of test cases to become unusable, answering question **Q3**.

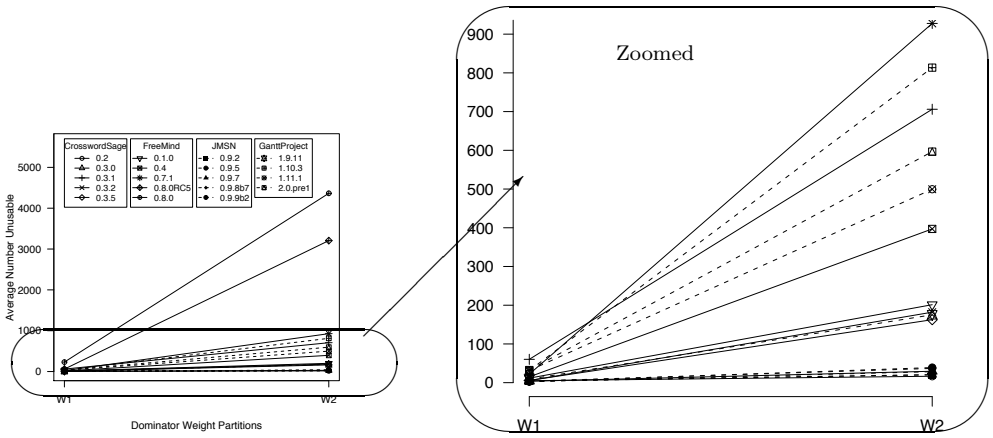


Fig. 15. Affect of Dominator Weight on Test Case Unusability

4.6 Addressing **Q4**: Studying the Effectiveness of the Checker and Repairer

Questions **Q1** through **Q3** were adequately answered by using data collected in the three steps outlined in Section 4.1; these three steps comprise of *Scenario 1*. The total time required for Scenario 1 is shown in Table III under column `Total Time`. Question **Q4**, on the other hand, requires studying the advantage, in terms of time, of using the checker and repairer, *i.e.*, how much time would a tester, who does not have access to these two tools, spend doing regression testing? In this section supplemental test cases are obtained *without* using the repairer, and a new process to detect unusable test cases *without* the checker is used. Scenario 2 excludes the repairer; Scenario 3 excludes the checker; Scenario 4 excludes both the checker and repairer.

Because Scenario 2 does not employ the repairer, no repaired test cases are available. In order to satisfy the event and event-interaction coverage criteria, supplemental test cases are generated by using the capture/replay tool. The number of supplemental test cases is shown in the last column of Table IV. Note that as discussed earlier, no test cases were obtained for Versions 0.7.1 and 0.8.0RC5 of FreeMind. Column `Supplemental Test Cases` of Table III shows the total time required to obtain the test cases. Figure 16 shows the time distributions by versions. The regression test suite obtained from Scenario 2 consists of all the test cases obtained by the capture/replay tool. The last column of Table III shows the total time needed for this scenario. The time is significantly more than that of Scenario 1 because of the manual test-case generation.

Because Scenario 3 does not employ our EFG-based automated checker, the checking of the unusability of each test case had to be done individually by actually executing it on the corresponding application version. A new test-case monitor was developed to check for "blocked" test cases; unusable test cases were blocked because, at some point during their execution, the test-case replayer stopped due to the unavailability of an event; the replayer performs an event as soon as it is available for execution. The new monitor was designed to terminate a test case's

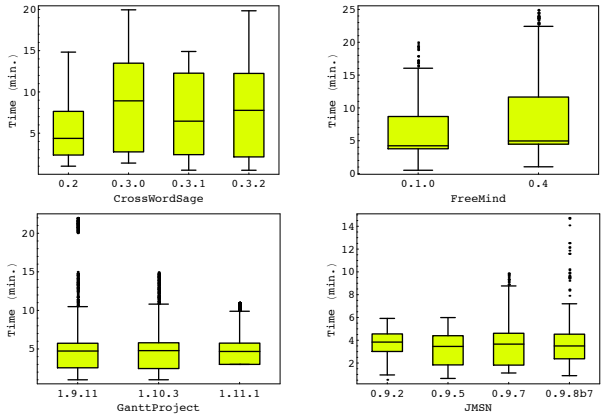


Fig. 16. Time Taken to Record Each Supplemental Test Case.

execution if an event was not available within a pre-specified time period. Previous experience with GUI testing overhead has shown that 20-30 seconds are a typical upper-bound for desktop GUI applications to respond to events and completely re-render the GUI (most events take a few seconds) [Xie and Memon 2007]. Hence, to be safe, the time-out period was set to 30 seconds. Reducing this time is certain to improve the performance of the monitor. However, this reduction also yields several false positives; before running this empirical study, several timeout limits were tried; even at a 20 second timeout limit, several test cases were terminated prematurely by the monitor and incorrectly marked as unusable; they would have executed to completion successfully if they had been given sufficient time to repaint all their widgets. With a 30 second limit, there were no false positives; the set of test cases deemed unusable by the EFG-based automated checker and the set computed by the new monitor was always identical, providing a good sanity check. The total time taken for this execution is shown in column `Computing Unusable Test Cases w/o Checker`; individual time distributions are shown by version in Figure 17. It is easy to see that the time taken by the replayer is much more than that of our checker. The last column of Table III shows the total time needed for this scenario. The time is slightly more than that of Scenario 1.

Finally, as expected, Scenario 4, with no checker or repairer, is the most expensive. The last 4 lines of Table III show the activities that were involved in each scenario. Figure 18 summarizes the total time for each scenario on each version.

To answer question **Q4**, *i.e.*, whether the checker and/or repairer helped to save a statistically significant amount of regression test-development time, an ANOVA test (with Alpha = 0.05) was conducted. The null hypothesis was that the means of all scenario data sets are equal; the alternative hypothesis was that the means are not equal. The observed p-value was 0.046; hence, the null hypothesis was rejected and the alternative hypothesis was accepted. The Tukey *post hoc* means comparison test showed that the differences between each scenario pair were also statistically significant. This analysis lead to the conclusion that the checker and/or repairer helped to save a statistically significant amount of regression test-development time,

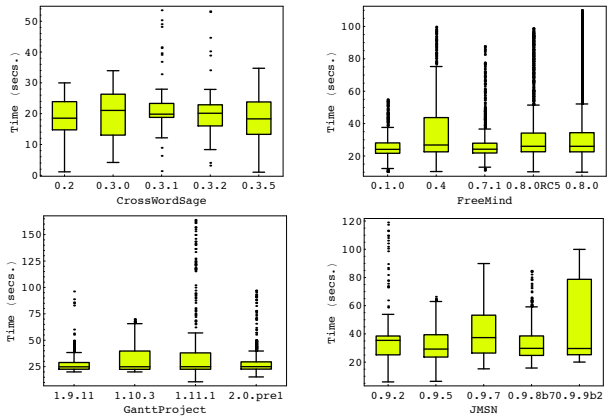


Fig. 17. Time Taken to Check Each Test Case Without Checker.

which, together with the data plots, answer question **Q4**.

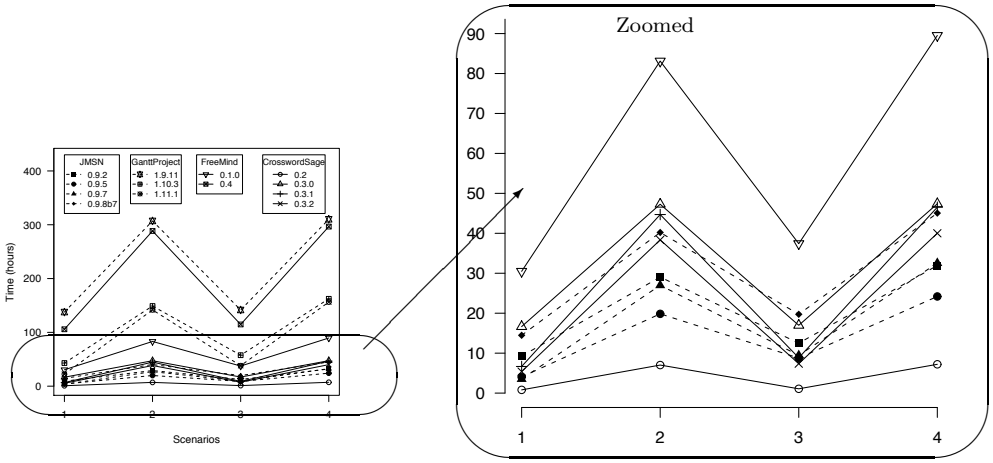


Fig. 18. Time Variation by Scenarios

5. CONCLUSIONS AND FUTURE WORK

This paper presented a new regression testing technique for GUIs that detects and repairs unusable test cases in a test suite. Developing GUI test suites is very time consuming and tedious; hence, the motivation for this work is to try to maintain the suites rather than create new ones. A representation called an event-flow graph was used to model the events of a GUI. This representation of the original and modified GUIs was compared to detect unusable test cases in the suite and then used to repair them. An empirical study showed for four widely-used open-source applications that (1) the repairing technique is effective in that many of the test

cases can be repaired, and is practical in terms of its time performance, (2) certain types of test cases are more prone to becoming unusable, and (3) certain types of dominator events, when modified, make a large number of test cases unusable.

Short-term future work stems from the threats to validity of the presented empirical study. Since GUITAR contains several techniques for test-case generation, multiple types of test cases will be generated and the impact of GUI changes on them will be studied. The goal is to derive a more comprehensive set of test case characteristics that make them unusable. A related issue is to generate test cases using different adequacy criteria [Memon et al. 2001] and study how they are impacted by changes. The GUI ripper will be extended to handle GUIs other than ones implemented in Java Swing, thereby allowing the study of a wider range of software. This empirical study used only four simple transformations. The test cases that were not repaired will be studied and new transformations will be developed that make these test cases usable.

Medium-term future work involves studying the effect of GUI changes on test oracle information (*i.e.*, expected output). GUITAR already contains mechanisms to regenerate the oracle information using GUI specifications, modeled as pre- and post-conditions [Memon et al. 2000]; it may be cheaper to reuse some of the existing information. The fault-detection effectiveness of the repaired test cases will also be studied. Finally, the repairing techniques currently pursue its objective by performing an exhaustive sequence of possible transformations. This approach will become expensive as the number and complexity of transformations increase. Search mechanisms (*e.g.*, genetic algorithms, AI Planning) may help to improve the approach's performance.

Long-term future work involves applying the repairing technique to non-GUI based software. For example, one way to test object-oriented software is to generate sequences of methods as test cases; during their execution, contract violations [Pacheco et al. 2007] may be checked. Moreover, although this research has been presented using EFG, it is applicable to state-machine models. Indeed a state machine model that is equivalent to an EFG can be constructed – the state would capture the possible events that can be executed on the GUI at any instant; transitions cause state changes whenever the number and type of available events change. Finally, this work will be extended to the general class of event-driven software.

Acknowledgments

The author thanks the anonymous reviewers of the original conference paper [Memon and Soffa 2003] whose comments and suggestions also helped to improve this paper's presentation. The anonymous reviewers of the Journal paper helped to extend the empirical study, reshape its results, and improve the flow of the text. Thanks also to Mary Lou Soffa, Adithya Nagarajan, Ishan Banerjee, Qing Xie, and Bin Gan who helped with the study, and to lay the foundation for this research. 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

Abbot 2003. Abbot Java GUI Test Framework. http://abbot.sourceforge.net.

Agrawal, H., Horgan, J. R., Krauser, E. W., and London, S. A. 1993. Incremental regression testing. In *Proceedings of the Conference on Software Maintenance*. IEEE Computer Society Press, Washington, 348–357.

Aho, A. V., Sethi, R., and Ullman, J. D. 1986. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA.

Beizer, B. 1990. *Software Testing Techniques*, 2nd ed. Van Nostrand Reinhold, New York.

Benedusi, P., Cimitile, A., and DeCarlini, U. 1988. Post-maintenance testing based on path change analysis. In *Proceedings of the IEEE Conference on Software Maintenance*. IEEE Computer Society Press, Washington, 352–368.

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

Hammontree, M. L., Hendrickson, J. J., and Hensley, B. W. 1992. Integrated data capture and analysis tools for research and testing an graphical user interfaces. In *Proceedings of the Conference on Human Factors in Computing Systems*. ACM Press, New York, NY, USA, 431–432.

Harrold, M. J., Gupta, R., and Soffa, M. L. 1993. A methodology for controlling the size of a test suite. *ACM Transactions of Software Engineering and Methodology 2,* 3 (July), 270–285.

Harrold, M. J., McGregor, J. D., and Fitzpatrick, K. J. 1992. Incremental testing of object-oriented class structures. In *Proceedings: 14th International Conference on Software Engineering*. ACM Press, New York, 68–80.

Harrold, M. J. and Soffa, M. L. 1989. Interprocedual data flow testing. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Testing, Analysis, and Verification (TAV3)*. ACM Press, New York, 158–167.

Hicinbothom, J. H. and Zachary, W. W. 1993. A tool for automatically generating transcripts of human-computer interaction. In *Proceedings of the Human Factors and Ergonomics Society 37th Annual Meeting*. SPECIAL SESSIONS: Demonstrations, vol. 2. ACM Press, New York, 1042.

JUnitResources 2005. JUnit, Testing Resources for Extreme Programming. http://junit.org/news/extension/gui/index.htm.

Kasik, D. J. and George, H. G. 1996. Toward automatic generation of novice user test scripts. In *Proceedings of the Conference on Human Factors in Computing Systems : Common Ground*. ACM Press, New York, 244–251.

Kepple, L. R. 1992. A new paradigm for cross-platform automated GUI testing. *The X Resource 3,* 1 (June), 155–178.

Kepple, L. R. 1994. The black art of GUI testing. *Dr. Dobb's Journal of Software Tools 19,* 2 (Feb.), 40.

Kung, D. C., Gao, J., Hsia, P., Toyoshima, Y., and Chen, C. 1996. On regression testing of object-oriented programs. *The Journal of Systems and Software 32,* 1 (Jan.), 21–31.

Lengauer, T. and Tarjan, R. E. 1979. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst. 1,* 1, 121–141.

Marick, B. 2002. Bypassing the GUI. *Software Testing and Quality Engineering Magazine 2*, 41–47.

Memon, A., Banerjee, I., and Nagarajan, A. 2003. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*. IEEE Computer Society, Washington, DC, USA, 260–269.

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

Memon, A. M. 2001. A comprehensive framework for testing graphical user interfaces. Ph.D. thesis, Department of Computer Science, University of Pittsburgh.

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

Memon, A. M., Pollack, M. E., and Soffa, M. L. 2000. Automated test oracles for GUIs. In *Proceedings of the ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering (FSE-8)*. ACM Press, New York, 30–39.

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

Memon, A. M. and Soffa, M. L. 2003. Regression testing of GUIs. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM Press, New York, NY, USA, 118–127.

Memon, A. M., Soffa, M. L., and Pollack, M. E. 2001. Coverage criteria for GUI testing. In *Proceedings of the 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)*. ACM Press, New York, 256–267.

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

Myers, B. A. 1995. User interface software tools. *ACM Transactions on Computer-Human Interaction 2,* 1, 64–103.

Onoma, A. K., Tsai, W.-T., Poonawala, M., and Suganuma, H. 1998. Regression testing in an industrial environment. *Commun. ACM 41,* 5, 81–86.

Ostrand, T., Anodide, A., Foster, H., and Goradia, T. 1998. A visual test development environment for GUI systems. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA-98)*. ACM Press, New York, 82–92.

Pacheco, C., Lahiri, S. K., Ernst, M. D., and Ball, T. 2007. Feedback-directed random test generation. In *ICSE'07, Proceedings of the 29th International Conference on Software Engineering*. ACM Press, Minneapolis, MN, USA.

Pollock, L. and Soffa, M. L. 1992. Incremental global reoptimization of programs. *ACM Transactions on Programming Languages and Systems 14,* 2 (Apr.), 173–200.

RationalRobot 2003. Rational Robot. http://www.rational.com.ar/tools/robot.html.

Rosenberg, D. 1993. User interface prototyping paradigms in the 90's. In *Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems – Adjunct Proceedings*. Tutorials. ACM Press, New York, 231.

Rosenblum, D. and Rothermel, G. 1997. A comparative study of regression test selection techniques. In *Proceedings of the IEEE Computer Society 2nd International Workshop on Empirical Studies of Software maintenance*. IEEE Computer Society Press, Washington, 89–94.

Rosenblum, D. S. 1995. A practical approach to programming with assertions. *IEEE Trans. Softw. Eng. 21,* 1, 19–31.

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

Rothermel, G. and Harrold, M. J. 1993. A safe, efficient algorithm for regression test selection. In *Proceedings of the Conference on Software Maintenance*. IEEE Computer Society Press, Washington, 358–369.

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

Rothermel, G. and Harrold, M. J. 1998. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering 24,* 6 (June), 401–419.

Rothermel, G., Harrold, M. J., Ostrin, J., and Hong, C. 1998. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proceedings; International Conference on Software Maintenance*. IEEE Computer Society Press, Washington, 34–43.

Shehady, R. K. and Siewiorek, D. P. 1997. A method to automate user interface testing using variable finite state machines. In *Proceedings of The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing (FTCS'97)*. IEEE Press, Washington - Brussels - Tokyo, 80–88.

Software Research, Inc., Capture-Replay Tool 2003. Software Research, Inc., capture-Replay Tool. Available at `http://soft.com`.

THATCHER, J. 1994. Screen reader/2–programmed access to the gui. In *ICCHP '94: Proceedings of the 4th international conference on Computers for handicapped persons*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 76–88.

VOAS, J. 1997. How assertions can increase test effectiveness. *IEEE Software 14,* 2, 118–119,122.

WALWORTH, A. 1997. Java GUI testing. *Dr. Dobb's Journal of Software Tools 22,* 2 (Feb.), 30, 32, 34.

WHITE, L. 1996. Regression testing of GUI event interactions. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society Press, Washington, 350–358.

WHITE, L. AND ALMEZEN, H. 2000. Generating test cases for GUI responsibilities using complete interaction sequences. In *Proceedings of the International Symposium on Software Reliability Engineering*. IEEE Computer Society Press, Washington, 110–121.

WHITE, L., ALMEZEN, H., AND SASTRY, S. 2003. Firewall regression testing of gui sequences and their interactions. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, Washington, DC, USA, 398.

WHITE, L., JABER, K., AND ROBINSON, B. 2005. Utilization of extended firewall for object-oriented regression testing. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE Computer Society, Washington, DC, USA, 695–698.

WinRunner 2003. Mercury Interactive WinRunner. http://www.mercuryinteractive.com/products/winrunner.

WITTEL, JR., W. I. AND LEWIS, T. G. 1991. Integrating the MVC paradigm into an object-oriented framework to accelerate GUI application development. Tech. Rep. 91-60-06, Department of Computer Science, Oregon State University.

WOHLIN, C., RUNESON, P., HOST, M., OHLSSON, M. C., REGNELL, B., AND WESSLEN, A. 2000. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, Norwell, MA, USA.

XIE, Q. AND MEMON, A. M. 2006. Model-based testing of community-driven open-source GUI applications. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance*. IEEE Computer Society, Los Alamitos, CA, USA, 145–154.

XIE, Q. AND MEMON, A. M. 2007. Designing and comparing automated test oracles for GUI-based software applications. *ACM Transactions on Software Testing and Methodology 16,* 1, 4.

YUAN, X. AND MEMON, A. M. 2007. Using GUI run-time state as feedback to generate test cases. In *ICSE'07, Proceedings of the 29th International Conference on Software Engineering*. ACM Press, Minneapolis, MN, USA.