Regression Testing of GUIs

Atif M. Memon
Dept. of Computer Science
University of Maryland
& Fraunhofer Center Maryland
College Park, MD 20742
atif@cs.umd.edu

Mary Lou Soffa*
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
soffa@cs.pitt.edu

ABSTRACT

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 are either reusable or unusable on the modified GUI. Since GUI test case generation is expensive, our goal is to make the unusable test cases usable. The idea of reusing these unusable (*a.k.a. obsolete*) test cases has not been explored before. In this paper, we show that for GUIs, the unusability of a large number of test cases is a serious problem. We present a novel GUI regression testing technique that first automatically determines the usable and unusable test cases from a test suite after a GUI modification. It then determines which of the unusable test cases can be repaired so they can execute on the modified GUI. The last step is to repair the test cases. Our technique is integrated into a GUI testing framework that, given a test case, automatically executes it on the GUI. We implemented our regression testing technique and demonstrate for two case studies that our approach is effective in that many of the test cases can be repaired, and is practical in terms of its time performance.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*

General Terms

Verification, reliability, human factors.

Keywords

Regression testing, repairing test cases, GUI testing, GUI control-flow graph, GUI call-graph, call-tree, classification of events.

*Partially supported by a grant from the National Science Foundation (CCR 9808590 and EIA 9806525) to the University of Pittsburgh.

ESEC/FSE'03, September 1–5, 2003, Helsinki, Finland.

1. INTRODUCTION

Graphical User Interfaces (GUIs) are pervasive in today's software systems and constitute as much as half of software code [18, 13]. The correctness of a software system's GUI is paramount in ensuring the correct operation of the overall software system. One way, and the common way, to gain confidence in a GUI's correctness is through comprehensive testing. GUI testing requires that test cases (sequences of GUI *events* that exercise GUI *widgets*) be generated and executed on the GUI [14]. However, currently available techniques for obtaining GUI test cases are resource intensive, requiring significant human intervention. Even though a few automated GUI test case generation schemes have been proposed [16], in practice, test cases are still being generated manually using *capture/replay tools* [11].

When using a capture/replay tool, a human tester interacts with the *application under test* (AUT); the capture component of the tool stores this interaction in a file that can be replayed later using the replay component of the tool. Our experience has shown that generating a typical test case with 50 events for different widgets takes 20-30 minutes using capture-replay tools. Because of the time that it takes, a tester typically develops few test cases (100-300) for the software interface and hence, each test case is valuable.

In addition, most GUIs are designed using *rapid prototyping* [18], in which software is modified and tested on a continuous basis. The continuous modification of a GUI requires that test cases be reusable across versions, as it would be too expensive to generate new test cases for each version.

Although regression testing [6, 22, 24, 25] is an important software maintenance activity for traditional software, accounting for as much as one-third of the total cost of software production [20, 4], regression testing of GUIs has remained a largely unexplored area. The needs of GUI regression testing differ from those of traditional software. Regression testing research has focused on the development of regression test selection techniques that from a test suite choose a set of test cases that represent correct input and are deemed necessary to validate the modified software. Test cases that cannot be rerun (also known as *obsolete test cases* [24]) are ignored and simply discarded. The problem of unusable test cases is especially serious for GUIs, since GUI modifications make a large number of test cases unusable, requiring expensive regeneration.

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 are still valid for the modified GUI and can be rerun. 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. In early capture/replay tools that represented user events in terms of pixel coordinates of GUI widgets (also known as *analog mode* [1]), a moved widget could not be identified. However, modern capture/replay tools do not rely solely on coordinates for test case execution but maintain extra information such as the handle, type, and label (if any) of the widget, enabling the replayer to locate the widget when it has been moved. However, even with these modern tools, a large number of test cases are made 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. As will be seen in Section 6 we demonstrate that more than 74% of the test cases become unusable.

In this paper, we present a novel regression testing technique for GUIs. Our key idea is not to throw away test cases that are unusable for the modified GUI but to automatically repair them so they can execute on the modified GUI. For reasons discussed later, we focus on producing test cases that are "similar" to the original ones. With our new repairing technique, a tester can (1) rerun test cases that are usable for the modified GUI, as currently done, (2) repair and rerun previously unusable test cases, and (3) create new test cases to test new functionality. As will be seen in Section 6, our technique was able to repair more than 70% of the test cases made unusable by a new version of the GUI in our case study.

To perform GUI regression testing, we leverage the GUI representations that we developed in our GUI testing framework [17, 15, 13]. More specifically, we create a control model of the event structure of a GUI, use the model to determine the modifications, check if the test is usable on the modified GUI and if not, repair it if possible. When there are multiple ways to repair a test case, we use all of the ways, and thus produce more test cases. We have developed tools for automatic creation of our regression techniques, the representations, and replay of the repaired test cases.[1]

The contributions of this paper include:

1. the first regression testing technique that automatically generates 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. a repairer that takes a repairable GUI test case and repairs it to execute on the modified GUI,
4. experimental demonstration on real applications that our technique is effective and practical, and
5. integration of the regression testing technique into a framework for GUI testing.

In the next section, we formally define a GUI test case and outline the different ways in which GUI modifications may effect GUI test cases. We then define GUI control-flow graphs and GUI call-graphs in Section 3. A GUI regression testing example is presented in Section 4. The design of a regression tester that employs the repairing technique is described in detail in Section 5. In Section 6, we describe results of regression testing case studies performed on Adobe's Acrobat Reader and WordPad. Finally, in Section 7, we present related research and in Section 8, conclude with a discussion of ongoing and future work.

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

2. GUI MODEL AND TEST CASES

In this section, we first present a model of GUIs that we developed for a GUI testing framework [17, 15, 13]. We then define a GUI test case and formally define unusable and usable test cases.

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 all the objects that it contains, and the values of all their properties. Formally we define the state of a GUI as follows:

Definition: State of a GUI is the set P of all the properties of all the objects O that the GUI contains.

A distinguished set of states called its *valid initial state set* is associated with each GUI.

Definition: A set of states S_I is called the **valid initial state set** for a particular GUI iff the GUI may be in any state $S_i \in S_I$ when it is first invoked.

The state of a GUI is not static; events performed on the GUI change its state. These states of a GUI are called *reachable states*. The events are modeled as state transducers.

Definition: The **events** $E = \{e_1, e_2, \ldots, e_n\}$ associated with a GUI are functions from one state to another state of the GUI.

The function notation $S_j = e(S_i)$ is used to denote that S_j is the state resulting from the execution of event e in state S_i. Events occur as part of a sequence of events. Of importance to testers are sequences that are permitted by the structure of the GUI. We restrict our testing to such *legal* event sequences, defined as follows:

Definition: A **legal event sequence** of a GUI is $e_1; e_2; e_3; \ldots; e_n$ where e_{i+1} can be performed *immediately* after e_i.

An event sequence that is not legal is called an *illegal* event sequence. For example, 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, we define a GUI test case as:

Definition: GUI test case T is a pair (S_0, e_1; e_2; ...; 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 not reachable in the GUI and/or its event sequence is illegal, then the test case is not executable.

Definition: 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.

Definition: GUI test case (S_0, $e_1; e_2; \ldots; e_n$) is **usable** if it can execute to completion on a modified GUI.

Since GUI test cases are expensive to develop, we have developed a new technique to repair them. We use a repre-

sentation of the structure of the GUI to first detect changes to the GUI's structure and then use this information to repair the unusable test cases. Also, since today's GUIs are large, i.e., they consist of a large number of events and windows, our representation decomposes the GUI into manageable parts.

3. REPRESENTATION

In this section we give an overview of a GUI control-flow graph (G-CFG) and a GUI call-graph (G-call graph) that we developed for a GUI testing framework [17]. In the framework, the represesentations are used for coverage and test oracles. We now develop a technique based on this representation to detect modifications to the structure of the GUI. These graphs together represent a GUI's structure and can be automatically obtained from the GUI by performing a traversal of the GUI's event structure [17].

By representing a GUI with G-CFGs and a G-call graph, the original and modified representations can be compared to reveal modifications made to the GUI and to identify unusable test cases. Also, since these representations decompose the GUI hierarchically into manageable *GUI components*, the repair can focus on only one component at a time.

GUIs, by their very nature, are hierarchical, and the hierarchy can be exploited to identify groups of GUI events that can be analyzed in isolation of other parts of the GUI. One hierarchy of the GUI, and the one used in this research, is obtained by examining *modal windows* in a GUI. A modal window is a window that, once invoked, monopolizes the GUI interaction, restricting the focus of the user to a specific range of events within the window until the window is explicitly terminated. The `Print` window in Adobe Acrobat Reader is an example of a modal window. All other windows in a GUI are called *modeless*[2] *windows* as they do not restrict the user's focus but they merely expand the set of GUI events available to the user. For example, in Adobe Acrobat Reader, performing the event `Find` opens a modeless window entitled `Find`.

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

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

Note that, by definition, events within a component do not interleave with events in other components without the components being explicitly invoked or terminated. Thus, the operation of modal windows is very much like procedure/method calls.

A GUI component is represented as a GUI control-flow graph (G-CFG). Intuitively, a G-CFG statically represents all possible interactions among the events in a component, just like a control flow graph for programs statically represents all possible program paths.

[2] http://java.sun.com/products/jlf/ed2/book/HIG.Glossary.html#51680

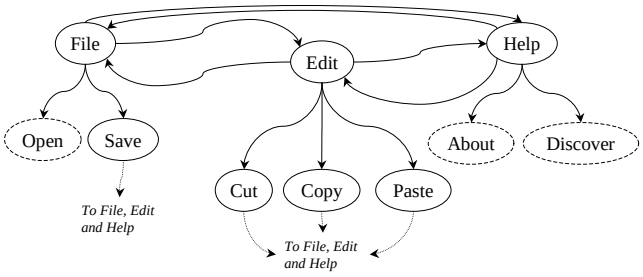


Figure 1: G-CFG for Part of Acrobat Reader.

Definition: A **G-CFG** for a component C is a 4-tuple <**V**, **E**, **B**, **I**> where:

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

An example of a G-CFG for a part of the `Main`[3] component of Acrobat Reader is shown in Figure 1. At the top are three vertices (`File`, `Edit`, and `Help`) that represent part of the pull-down menu of Acrobat Reader. They are events that are available when the `Main` component is first invoked. Once `File` has been performed in Reader, any of `Edit`, `Help`, `Open`, and `Save` events may be performed. Hence there are edges in the G-CFG from `File` to each of these events. Note that `Open`, `About` and `Discover` are shown with dashed ovals. We use this notation for events that invoke other components, i.e., **I** = {`Open`, `About`, `Discover`}. Other events include `Save`, `Cut`, `Copy`, and `Paste`. After any of these events is performed in Reader, the user may perform `File`, `Edit`, or `Help`, shown as edges in the G-CFG.

Once all the components of the GUI have been individually represented as G-CFGs, the remaining step is to construct a G-call graph to identify interactions among components. These interactions take the form of invocations, defined formally as:

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

Intuitively, a G-call graph shows the invokes relationships among all the components in a GUI. In general, the relationships among components are represented by a directed acyclic graph (DAG), since multiple components may invoke a component. However, the DAG can be converted into a tree by copying nodes. A tree model simplifies our algorithms based on tree traversals of the call-tree. Formally, a call-tree is defined as:

Definition: A GUI **call-tree** is a triple $< \mathcal{N}, \mathcal{R}, \mathcal{B} >$, where $\mathcal{N}$ is the set of components in the GUI and $\mathcal{R} \in \mathcal{N}$

[3] Without loss of generality, we assume that all GUIs have a `Main` component, i.e., the component that is presented to the user when the GUI is first invoked.

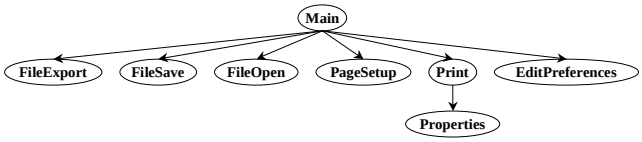


Figure 2: GUI call-tree for Part of Acrobat Reader.

is a designated component called the `Main` component. $\mathcal{B}$ is the set of directed edges representing the invokes relation between components, i.e., $(C_x, C_y) \in \mathcal{B}$ iff C_x **invokes** C_y.

Figure 2 shows an example of a GUI call-tree representing a part of the Acrobat Reader's GUI. The nodes represent the components of the GUI and the edges represent the invokes relationship between the components. Component names indicate their functionality. For example, `FileOpen` is the component of Reader used to open files. The tree in Figure 2 has an edge from `Main` to `FileOpen` showing that `Main` contains an event, namely `Open` (see Figure 1) that invokes `FileOpen`.

4. A GUI REGRESSION TESTING EXAMPLE

We now present an overview of our GUI regression testing technique by showing (1) examples of GUI modifications, (2) examples of test cases that have become unusable for the modified GUI, (3) an intuitive idea of how analysis of the GUI can help identify the unusable test cases, and (4) how unusable test cases may be repaired to obtain new test cases. It should be noted that only structural changes in a GUI make a test case unusable. If the semantics of a node change, then the test cases would be usable although the GUI's output may change.

Figure 3 presents a GUI, its modified version, and their corresponding G-CFGs. The original GUI consists of 4 events, `Cut, Copy, Paste`, and `Print`, all directly accessible when the GUI is invoked. The modified GUI contains 3 of the 4 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 have not changed. Figures 3(c) and (d) show the G-CFGs of the original and modified GUIs respectively. The original GUI's G-CFG is fully connected with 4 vertices representing the 4 events. The modified GUI's G-CFG 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. The following four sets of changes may be obtained, summarizing the differences between the two G-CFGs:

1. **events_deleted** = {`Print`}.
2. **events_added** = {`Edit`}.
3. **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)`}.
4. **efg_edges_added** = {`(Edit, Edit)`, `(Edit, Cut)`, `(Edit, Copy)`, `(Edit, Paste)`, `(Cut, Edit)`, `(Copy, Edit)`, `(Paste, Edit)`}.

Four event sequences used to test the original GUI are shown in Table 1. Column 1 shows the test case number, column 2 shows the event sequence of the test case, column 3 shows the events in the G-CFG used by the test case, and column 4 shows the edges of the G-CFG covered by the test case. The following observations can be made by examining these test cases and the 4 sets above:

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

Intuitively, looking at the original and modified GUIs, event sequences 3 and 4 may be modified (or *repaired*) to obtain legal event sequences. One repair to event sequence 3 yields <`Cut; Edit; Paste`> and two repairs 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. It is not obvious how event sequence 1 may be repaired since it contains an event, namely `Print`, that is no longer available in the modified GUI. In this example, 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. After repairing, the test designer can choose from a total of three event sequences and use them for regression testing. Note that since 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 plus any new test cases.

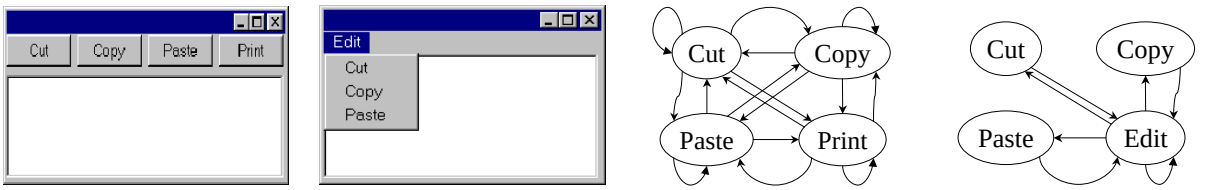
Since a test case may become unusable by several modifications made to the GUI, it may need to be repaired several times before it is usable, and our techique performs multiple actions to repair a test case.

Note that only adding new events and edges and not deleting any edges or nodes from a G-CFG cannot result in illegal event sequences. The event sequences from the original test suite neither use any of the new events nor do they cover any of the new edges. Since edges and nodes are not deleted, the test case can still exercise the GUI.

5. REGRESSION TESTING

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 could be merged together in an implementation.

The regression tester takes as input the G-CFGs and G-call trees for 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. Intuitively, a test case can be repaired if its initial state is still valid for the modified GUI (i.e., the GUI can be brought into the state) and if its event sequence can be made



(a) The Original GUI. (b) The Modified GUI. (c) The Original G-CFG. (d) The Modified G-CFG.

Figure 3: A Regression Testing Example.

#	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 1: Four Event Sequences for the Original GUI.

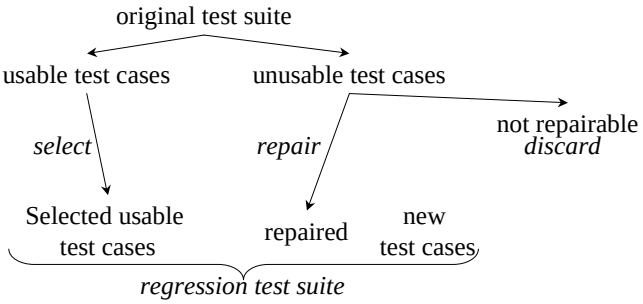


Figure 4: The New Regression Testing Method.

legal for the modified GUI. To 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 [3]. This sequence can be found by skipping over events or by including events from the modified GUI. Formally, we define a repairable test case as:

Definition: An unusable test case is **repairable** if its initial state S_0 is reachable, and its event sequence can be made legal for the modified GUI, i.e., $(e_i, e_{k>i}) \in \mathbf{E}$) or $\{(e_i, e_x), (e_x, e_{k>i})\} \subset \mathbf{E}$ for $\mathbf{E}$, for $1 \leq i \leq n$ of the modified GUI's G-CGF.

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 4. Note that new test cases, generated to test those unusable parts of the GUI that were not tested by the repaired test cases, are also a part of the regression test suite plus selected usable test cases.

The regression tester, contains the following components:

- **Test case checker** partitions the original test suite into (1) usable test cases, (2) repairable unusable test cases, and (3) non-repairable unusable test cases.
- **Test case repairer** repairs the unusable test cases by adding and deleting events to the test case in order to match the event sequence with the modified GUI.

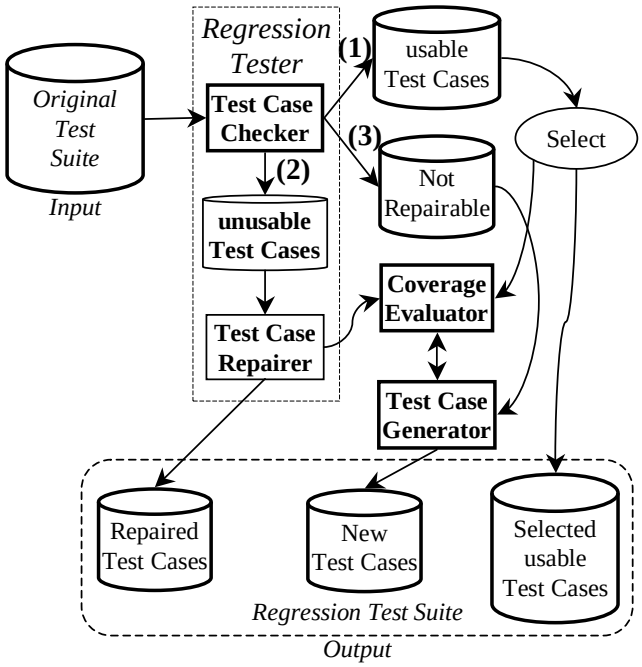


Figure 5: Regression Tester's Components and their Interactions.

Figure 5 shows the checker and repairer of the regression tester and their interactions. The figure also shows the interactions of these components with a **test case generator** and a **coverage evaluator** to generate new test cases that test new parts of the GUI [16, 17]. Together, the repaired, new, and selected usable test cases form the regression test suite. We now present the details of the design of the test case checker and repairer.

5.1 Test Case Checker

The test case checker's primary function is to identify unusable test cases and of those, which can be repaired. The test-case checker determines whether an event sequence in

a test case is reusable by first identifying the modifications made to the GUI by traversing the G-CFGs and G-call trees of the original GUI amd modified GUI.

Since we want this analysis to be efficient, we simplify the analysis by making a number of reasonable assumptions. We assume that events and components: (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 components 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 G-CFG and G-call tree. Note that similar approaches of tracking additions/deletions on control flow graphs for software have been used for incremental data-flow and code-optimizations [19] and incremental testing [9].

Events within a component, represented by a G-CFG, may be modified by adding or deleting a vertex or edge. If $GCFG_o$ and $GCFG_m$ are the G-CFGs of a component that exists in both the original GUI and the modified GUI respectively, then the following sets of modifications are obtained by performing set subtraction. Note that the functions $Vertices$ and $Edges$ return the sets $\mathbf{V}$ (the set of vertices) and $\mathbf{E}$ (the set of edges) for the G-CFG in question.

1. The set of all new vertices in the G-CFG: **vertices_added** $\leftarrow Vertices(GCFG_m) - Vertices(GCFG_o)$;
2. The set of all vertices deleted from the original G-CFG: **vertices_deleted** $\leftarrow Vertices(GCFG_o) - Vertices(GCFG_m)$;
3. The set of all new edges added to the G-CFG: **efg_edges_added** $\leftarrow Edges(GCFG_m) - Edges(GCFG_o)$;
4. The set of edges deleted from the original G-CFG: **efg_edges_deleted** $\leftarrow Edges(GCFG_o) - Edges(GCFG_m)$;

As illustrated earlier in Section 4, the above sets can be used to identify unusable test cases.

Similarly, a call-tree may be modified by adding or deleting a component or edge. Let G_o and G_m be the call-trees of the original and modified GUIs respectively. The following sets of modifications may be obtained from these two call-trees. Note that $Nodes$ and $CompEdges$ return the sets $\mathcal{N}$ and $\mathcal{B}$ for the call-tree respectively.

1. The set of components added to the G-call tree: **components_added** $\leftarrow Nodes(G_m) - Nodes(G_o)$;
2. The set of components deleted from the G-call tree: **components_deleted** $\leftarrow Nodes(G_o) - Nodes(G_m)$;
3. The set of edges added to the G-Call tree: **comp_edges_added** $\leftarrow CompEdges(G_m) - CompEdges(G_o)$;
4. The set of edges deleted from the G-call tree: **comp_edges_deleted** $\leftarrow CompEdges(G_o) - CompEdges(G_m)$;

Note the difference between the edges of a G-CFG and those of a call-tree. Edges of a G-CFG are ordered pairs of the form (e_x, e_y), where e_x and e_y are events, whereas edges of the G-call tree are ordered pairs of the form (C_x, C_y), where C_x and C_y are components. An edge in the G-call tree (C_x, C_y) represents the set of all edges (e_y, e_z), where e_y is an event in component C_x that invokes C_y.

The set of modifications obtained above are used to identify unusable test cases. Specifically, the following two sets are used to identify unusable test cases:

1. **vertices_deleted**, and
2. **edges_deleted** $\longleftarrow$ **efg_edges_deleted** $\cup$ `EventEdges`(**comp_edges_deleted**), where EventEdges is a function that takes a set of G-call tree edges and returns the corresponding edges in terms of events.

To identify unusable test cases, we can employ at least two different techniques. The first one, which we call the *graph matching* technique, takes an event sequence $< e_1; e_2; \ldots; e_n >$ and determines whether $(e_i,\ e_{i+1})$ for $1 < i < n-1$ is a valid edge in a modified G-CFG. This technique is simple to implement and shows good performance for small test suites. An alternative technique, which we show in Section 6 to be more efficient than graph matching for large test suites, records GUI modifications in two *bit vectors*, EDGES-MODIFIED and EVENTS-MODIFIED. Each test case is also associated with two bit vectors, EVENTS-USED and EDGES-USED. Determining whether a test case is usable/unusable can be 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. Note that 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. As will be seen later, 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.

5.2 Test Case Repairer

The test case repairer repairs (algorithm given in Figure 6) illegal event sequences so that is can be executed on the modified GUI. An illegal event sequences uses either a deleted event or a deleted edge. Intuitively, if an event e_i, at position i in an event sequence is deleted from the GUI, then an *event-sequence repairer* must remove e_i from the event sequence. However, to obtain a legal resulting event sequence, the event-sequence repairer 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.

It could happen that there is more than one way to repair a test case. When multiple ways are found, then all of the repairs are used to produce more test cases.

Once such an e_j is found, then the sub-sequence $< e_i; \ldots; e_{j-1} >$ is deleted from the event sequence and, in case 2, e_x is inserted. Figure 7(a) shows these two cases. In case 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 case 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, this technique 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. However, computing such a sequence is expensive and can produce a

```
ALGORITHM : EventSeqRepairer(                                          1
  S: illegal event sequence; vertices_deleted: vertices;               2,3
  edges_deleted: edges; EVENTS: events;                                4,5
  EVENTS-USED, EDGES-USED: Bit vector)                                 6,7
{ FOREACH (e_i ∈ vertices_deleted) DO                                  8,9
    WHILE (e_i^th bit of EVENTS-USED == 1) DO                          10
      repairability ⟵ repair_del_event(t, e_i);                        11
      IF (! repairability) THEN RETURN(FALSE);                         12
      UPDATE(EVENTS-USED, S);                                          13
  UPDATE(EDGES-USED, S);                                               14
  FOREACH ((e_i, e_j) ∈ edges_deleted) DO                              15
    IF ((e_i ∈ EVENTS && e_j ∈ EVENTS) THEN                            16
      WHILE ((e_i, e_j)^th bit of EDGES-USED == 1) DO                  17
        repairability ⟵ repair_del_edge(S, (e_i, e_j));                18
        IF (! repairability) THEN RETURN(FALSE);                       19
        UPDATE(EDGES-USED, S);                                         20
  RETURN(TRUE); }                                                      21,22
PROCEDURE : repair_del_event(                                          23
  S: Event Sequence; e: Event)                                         24,25
{ FOR k ⟵ p+1 TO n DO                                                  26,27
    IF e_k ∈ follows(e_{p+1}) THEN                                     28
      S ⟵ < e_1; ...; e_{p-1}; e_k; ...; e_n >;                        29
      done1 ⟵ TRUE; break;                                             30
    ELSE IF ∃ e_x ((e_x ∈ follows(e_{p-1}))
          && (e_k ∈ follows(e_x))) THEN                                31
      S ⟵ < e_1; ...; e_{p-1}; e_x; e_k; ...; e_n >;                   32
      done2 ⟵ TRUE; break;                                             33
    RETURN (done1 || done2); }                                         34,35
PROCEDURE : repair_del_edge(                                           36
  S: Event Sequence; (e_a, e_b): Edge)                                 37,38
{ FOR k ⟵ b TO n DO                                                    39,40
    IF e_k ∈ follows(e_a) THEN                                         41
      S ⟵ < e_1; ...; e_a; e_k; ...; e_n >;                            42
      done1 ⟵ TRUE; break;                                             43
    ELSE IF ∃ e_x ((e_x ∈ follows(e_a))
          && (e_k ∈ follows(e_x))) THEN                                44
      S ⟵ < e_1; ...; e_a; e_x; e_k; ...; e_n >;                       45
      done2 ⟵ TRUE; break;                                             46
    RETURN (done1 || done2)}                                           47
```

Figure 6: Algorithm for the Event-sequence Repairer.

test case that is very different from the original one. Our goal is to produce similar test cases to the original one, a necessary condition to retest already tested functionality of the GUI.

Similarly, Figure 7(b) shows the repairing technique for the deleted edge (e_i, e_j). In this technique, the event sequence is scanned from left to right, starting with the event e_j, the second element in the deleted edge. Case 1 tries to find an event e_a from the subsequence $< e_j; \ldots; e_n >$ such that e_a `follows` e_i. Case 2 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.

As noted earlier, an event sequence may have become illegal because of several changes made to the GUI. Each event sequence is checked for all instances of deleted events and edges that made the event sequence illegal.

The algorithm for the repairer is shown in Figure 6. The main algorithm is called `EventSeqRepairer` that takes a number of parameters: (1) the illegal event sequence **S**, (2) the set **vertices_deleted**, (3) the set **edges_deleted**, (4) the set of all the **events** available in the modified GUI, (5) the bit vector **EVENTS-USED** associated with the event sequence, and (6) the bit vector **EDGES-USED**. `EventSeqRepairer` returns TRUE if the event sequence was repaired successfully, and FALSE otherwise. The algorithm starts by examining each event e_i that was deleted from the GUI (Line 9). If **S** uses this event (`Line 10`), then it is illegal. The procedure `repair_del_event` is invoked to repair **S** (`Line 11`). If **S** is repairable, then `repair_del_event` returns TRUE, otherwise `EventSeqRepairer` terminates with a FALSE result (`Line 12`). Since `repair_del_event` may have changed the events used by **S**, the bit vector **EVENTS-USED** is updated to reflect the changes (`Line 13`). Note that the WHILE loop continues examining the event sequence for the deleted event e_i. After **S** has been repaired for all deleted events, its **EDGES-USED** is updated to reflect all the changes made so far (`Line 14`). `EventSeqRepairer` continues by examining each edge (e_i, e_j) that was deleted (`Line 15`). It makes sure that both events e_i and e_j are available in the GUI (`Line 16`). If **S** uses this edge (`Line 17`), then it is illegal. The procedure `repair_del_edge` is invoked to repair **S** (`Line 18`). If **S** is repairable, then `repair_del_edge` returns TRUE, otherwise `EventSeqRepairer` terminates with a FALSE result (`Line 19`). **EDGES-USED** is updated to reflect the changes made to **S** (`Line 20`). If `EventSeqRepairer` has not terminated using any of the RETURN statements (`Lines 12, 19`), then the event sequence has been successfully repaired (`Line 21`).

The procedure `repair_del_event` tries to repair the illegal event sequence caused by deleting an event. It takes two parameters: (1) the event sequence **S**, and (2) the deleted event e. It starts scanning the subsequence $< e_{p+1}; \ldots; e_n >$ from left to right (`Line 27`) until one of the cases shown in Figure 7(a) is found or the sequence terminates. If case 1 is solved (`Line 28`), then the sequence is updated (`Line 29`) and success reported (`Line 30`). Otherwise if case 2 is solved (`Line 31`), then the sequence is updated (`Line 32, 33`). The procedure `repair_del_edge` is similar to `repair_del_event`. It scans the subsequence $< e_b; \ldots; e_n >$ from left to right until one of the cases of Figure 7(b) is found.

Note that since repairer employs information from the G-CFG and G-call trees (represented by `follows`), the event sequence repairer is guaranteed to produce legal event sequences.

6. CASE STUDIES

Having presented the algorithms of the regression tester, we now examine its practicality using actual software versions and test runs. We identified the following questions that need to be answered to show the practicality of the repairing process and to explore the cost of using different implementations of the checker.

1. How many test cases are made unusable by GUI modifications across versions?
2. How many unusable test cases are repairable?
3. How much time does the checker and repairing processes take?
4. How does the graph-matching based checker algorithm compare with the bit-vector based algorithm; for what test suite size does one outperform the other?

To answer the questions we needed to take different versions of software, count the number of test cases made un-

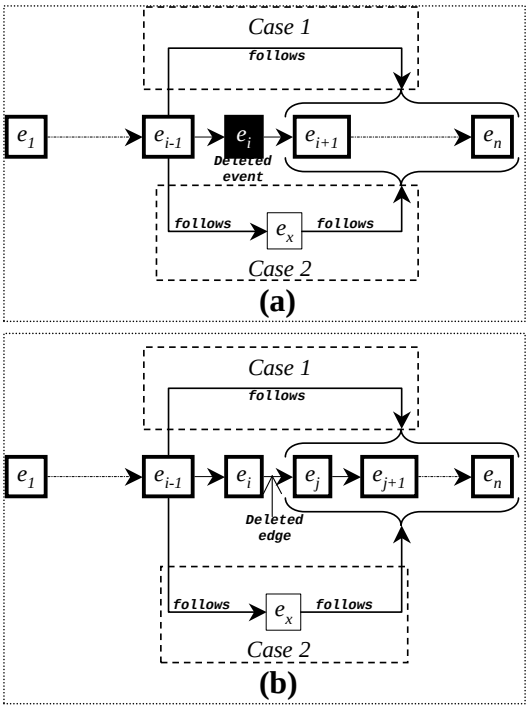


Figure 7: Repairing an Event Sequence that Uses a (a) Deleted Event e_i, and (b) Deleted Edge (e_i, e_j).

usable/usable and not repairable, while measuring the cost of the overall process.

6.1 Subject Applications

For our studies, we used two programs, each with two versions as our subjects. We used one commercially available software - Adobe's Acrobat Reader, and a program developed in-house - our own implementation of MS WordPad.

The first subject application, Adobe's Acrobat Reader version 4.0 (for Linux) and version 5.0 (for MS Windows 2000) were used as our original and modified GUIs respectively. Adobe's Acrobat Reader version 4.0 consists of 15 components with 176 events (not counting short-cuts). Version 5.0 is much more complex, consisting of 25 components with 351 events.

The second subject application was developed in-house. We employed our own specifications and implementation of the MS WordPad software. The software consists of 36 modal windows, and 362 events (not counting short-cuts). Our implementation of WordPad is similar to Microsoft's WordPad except for the **Help** menu, which we did not model. A modified version of the WordPad GUI was created by grouping three events into a separate pull-down menu. As Figure 8 shows, `Find`, `FindNext`, and `Replace` were grouped into a new pull-down menu item called `Search`. Note that these items were deleted from their original location, i.e., from the `Edit` menu. In terms of the elements of the G-CFGs of WordPad's `Main` component, no nodes were deleted but three edges were deleted – (`Edit`, `Find`), (`Edit`, `FindNext`), and (`Edit`, `Replace`). One node labeled `Search` was added with its associated edges.

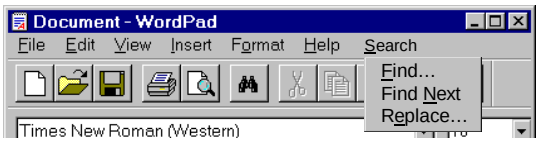


Figure 8: Modified WordPad GUI with a New Pull-down Menu Item called `Search`.

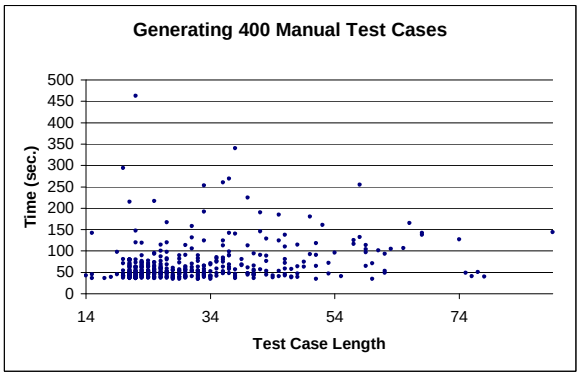


Figure 9: Time Taken to Generate 400 Test Cases.

6.2 Study 1: Performance and Effectiveness of the Regression Testing Technique

For both subjects, the G-CFGs and G-call trees of the original and modified GUIs were automatically generated. All the components of the regression tester were implemented. Specifically, the test case checker and repairer were implemented in Perl. The study was conducted on a 1.7 GHz Pentium Workstation with 1 GB of RAM. 400 test cases were generated manually using a capture/replay tool[4] for each subject application. We then executed the checker to determine how many of the test cases became unusable and how many were (un)repairable. We then repaired these test cases. Results for each subject application are described next.

Adobe's Acrobat Reader: We manually generated 400 test cases in 7.59 hours using the capture/replay tool. The time taken to generate each test case is shown in Figure 9. The changes in the GUI made 296 (74%) test cases unusable. The remaining 104 (26%) test cases were usable. The total time taken for the checker was 6.5 sec. We then executed the test case repairer on the unusable test cases and successfuly repaired 211 (71.3%) of them. The total time taken for repairer was 17.76 sec. We had a total of 315 (78.75%) usable test cases. Since the original test cases were generated manually and our repairing technique did not make significant changes to the test cases, the resulting test cases were similar to the original. The results are summarized in Figure 10 and Table 2.

WordPad: We manually generated 400 test cases using the capture/replay tool in 5.47 hours. The GUI modifications affected 210 of the 400 test cases. The checker took 6.15 seconds. We repaired all 210 test cases since all that was needed was to replace `Edit` with `Search` in each unus-

[4]Available at http://guitar.cs.umd.edu

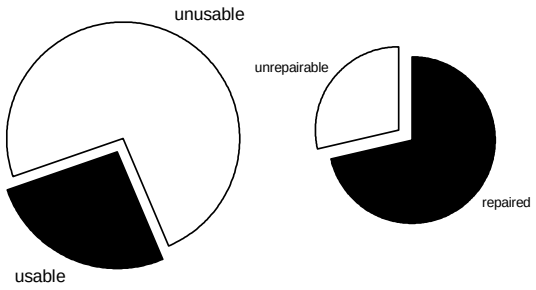


Figure 10: Results of Adobe's Reader Case Study.

Step	Subject Application	Time
Manual Generation	Reader	7.59 hrs.
	WordPad	5.47 hrs.
Checker	Reader	6.5 sec.
	WordPad	6.15 sec.
Repairer	Reader	17.76 sec.
	WordPad	18.01 sec.

Table 2: Time Taken at a Glance.

able test case. The lengths of the original and repaired test cases were identical. The time taken was 18.01 sec. These times are summarized in Table 2.

In this case study, we have demonstrated that our repairing technique is practical and effective and can be used for GUI regression testing. Our experience with GUI testing has shown that the currently employed techniques, which are largely manual aided with capture/replay tools, require a long time to develop only a few hundred test cases. The use of this technique helps reduce the cost of GUI regression testing. Note that the repaired test cases obtained form an important part of the regression testing test suite. The test designer will also need to generate additional test cases to check the new parts of the GUI.

6.3 Study 2: Comparing the Bit-vector with a Graph Matching Checker

As noted in Section 5.1, we can employ two different algorithms for our checker. In this study, we compare the execution times of the bit vector based checker with the simpler graph matching based checker. The time for the bit vector based checker consisted of measuring the setup time for (1) computing the sets of vertices/edges/components added/deleted (2) for each test case the bit vectors EVENTS EDGES-USED, and (3) two global vectors EVENTS-MODIFIED and EDGES-MODIFIED encoding the edges and events modified in the GUI. Once this information was computed, determining whether a test case was unusable was done by using the binary operation AND on each test case's bit vectors and the global EVENTS/EDGES-MODIFIED vectors. The time taken for this algorithm is shown by the curve marked **Bit-vector** in the graph of Figure 11. The time taken for the graph matching algorithm is shown by the curve marked **Graph** in the graph of Figure 11.

Figure 11 shows that although the bit-vector based checker

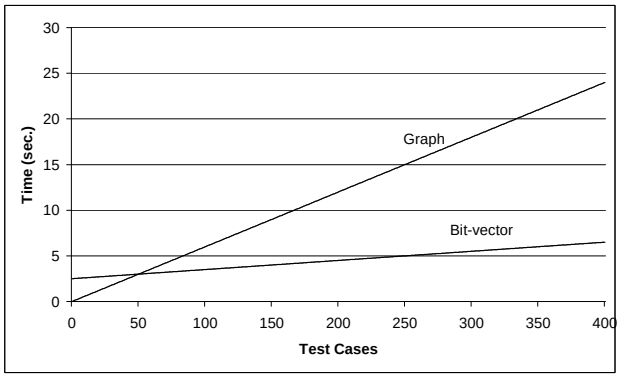


Figure 11: Comparing Bit-vector with Graph Matching Checker.

requires initial setup time, subsequent detection of unusable test cases is very fast. In fact, it performs better than the graph algorithm for more than 50 test cases. Also, the bit vectors computed for each test case can be used across multiple versions of the GUI.

7. RELATED WORK

Although regression testing of conventional software has received a lot of attention [6, 22, 24, 25], there has been almost no reported research on GUI regression testing. The only reported research was by White [27] who proposes a Latin square method to reduce the size of the regression test suite. The underlying assumption made therein was that it is enough to check pair wise interactions between components 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.

Record/playback tools [7] are currently the most popular tools for GUI testing but they provide little support for regression testing. Consequently, even a small change in the layout of the GUI requires redeveloping a large number of test cases from scratch.

Several strategies for regression testing of conventional software have been proposed [8, 21, 12]. One regression testing strategy proposes rerunning all test cases that have not become obsolete. Since this *retest-all strategy* is resource intensive, numerous efforts have been made to reduce its cost. *Selective retest techniques* [2, 5, 10] 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 [24], and (2) *coverage identification problem*, i.e., identifying portions of the software that require additional testing. Solutions to the regression test selection problem traditionally compare structural representations (e.g., control-flow graphs [24], control-dependence graphs [23]) 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* re-

quire the selection of every existing test case that exercises any program element that could be affected by a given program change. Although computationally less expensive than the retest-all strategy, safe approaches still make heavy demands on resources. At the other end of the spectrum of selective retest strategies are *minimization approaches* that attempt to select the smallest set of test cases necessary to test affected program elements at least once [26]. These techniques attempt to assure that some structural coverage criterion is met by the test cases that are selected. Practical strategies fall between safe and minimization strategies.

Other regression testing techniques include analyzing changes in functions, types, variables, and macro definitions [21], using def-use chains [8], constructing procedure dependence graphs [6], and analyzing code and class hierarchy for object-oriented programs [12].

8. CONCLUSIONS

This paper presents a new regression testing technique for GUIs that repairs unusable test cases. Test cases are very time consuming and tedious to construct manually so our motivation for this work is to try to maintain test cases rather than generate new ones. To represent the events of a GUI, we employed representations that showed the event behaviour of a component by a GUI control flow graph, and the invoking behavior of components by a GUI call-graph. These representations of the original and modified GUIs are compared to detect unusable test cases and then used to repair them. We show that our repairing technique is efficient and effective.

Modification of conventional software also produces obsolete test cases [24]. Studies need to be conducted to determine whether the repairing technique developed in this paper can be extended to repair unusable test cases for conventional software.

9. REFERENCES

[1] A. Khetawat. Collaborative Computing on the Internet. Master's thesis, Electrical and Computer Engineering, North Carolina State University, Raleigh, N.C., May 1997.

[2] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. A. London. Incremental regression testing. In *Proceedings of the Conference on Software Maintenance*, pages 348–357, Washington, Sept. 1993.

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

[4] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, 2nd edition, 1990.

[5] P. Benedusi, A. Cimitile, and U. DeCarlini. Post-maintenance testing based on path change analysis. In *Proceedings of the IEEE Conference on Software Maintenance*, pages 352–368, 1988.

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

[7] M. L. Hammontree, J. J. Hendrickson, and B. W. Hensley. 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*, pages 431–432, New York, NY, USA, May 1992.

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

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

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

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

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

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

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

[15] A. M. Memon, M. E. Pollack, and M. L. Soffa. Automated test oracles for GUIs. In *Proceedings of the ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering (FSE-8)*, pages 30–39, NY, Nov. 8–10 2000.

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

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

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

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

[20] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 1994.

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

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

[23] G. Rothermel and M. J. Harrold. A safe, efficient algorithm for regression test selection. In *Proceedings of the Conference on Software Maintenance*, pages 358–369, 1993.

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

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

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

[27] L. White. Regression testing of GUI event interactions. In *Proceedings of the International Conference on Software Maintenance*, pages 350–358, Washington, Nov.4–8 1996.