Test Case Generation and Reusing Test Cases for GUI Designed with HTML

Datchayani M
Department of Information Technology, Valliammai Engineering College, SRM Nagar, Kattankulathur, Chennai, India
m.dhaksh@gmail.com

Arockia Xavier Annie R [1], and Yogesh P [2]
[1] Department of Computer Science and Engineering, [2] Department of Information Science and Technology
College of Engineering, Anna University, Chennai, India
{annie, yogesh}@annauniv.edu

Benet Zacharias
Senior Consultant, Process Excellence, Wipro Consulting Services, Sholinganallur, Chennai - 600 119, India
benetzacharias@yahoo.co.uk

Abstract—**Graphical User Interface (GUI) is pervasive to the extent that half of the code of the software systems written today is to produce the required GUIs. Test case generation for GUI based software systems is complex as it is necessary to include all possible sequences of events that may be exercised by the widget or end-user. The major issue with GUI based systems is that even a single change in the GUI may make the entire suite of existing test cases unusable. Hence a solution to analyze the existing test cases (i.e. the test cases that were already generated before modification of GUI) and identify the test cases that have become unusable and that are reusable in the context of the modified GUI is provided here. Test cases that are reusable are retained and the test cases that have become unusable are subjected to transformations. These transformations make the unusable test cases reusable through the construction of Event Flow Graph (EFG) generated and parsing EFG using Depth First Search (DFS) to identify reusable and unusable test cases.**

Index Terms—**Graphical User Interface, Test Case Generation, Event Flow Graph, Depth First Search, Unusable Test Cases, Reusable Test Cases.**

I. Introduction

GUIs are pervasive in today's software systems and constitute as much as half of software code. The correctness of a software system's GUI is paramount in ensuring the correctness of the overall software system. The common way to ensure the GUI's correctness of the software systems is the comprehensive testing. Comprehensive testing includes all possible sequence of events that may be exercised by the widgets or the end users through the GUI of the software system. This testing requires both the generation and the execution of the above test cases. Techniques that are available at present for obtaining GUI test cases are resource intensive and require significant human intervention. Even though few automated GUI test case generation schemes like GUITAR (GUI Testing Framework) and DART (Daily Automated Regression Tester) [3] have been proposed, predominantly test cases are still being generated manually using capture/replay tools like QTP (Quick Test Professional) and WinRunner [5], [9].

Test case generation becomes complex when the software system undergoes a change. The software system may undergo a change either with respect to the GUI or with respect to the functionality Changes that occur in the GUI of the software system have more drastic effect on usability of a test case than the changes that occur in the functionality of the software system. Even a small change in the software system may make most of the existing test cases unusable for the modified software system [5].

The conventional capture/replay tools [1], show the interaction of the human tester with the application under test in a file and replays this file whenever required. From experiments, it is observed that generating a typical test case with 10 events for different widgets takes 20-30 minutes using capture-replay tools. It is very difficult for the human tester to generate the test cases for software systems that are designed using rapid prototyping that involves continuous modification and testing.

A. Objective of the System

An important goal of this work is to make the available test cases reusable for the modified version which would become unusable otherwise. Another major goal of this work is the test case generation for the acceptance testing of the GUI based software systems. To generate the test cases for the acceptance testing, it is necessary to capture all the sequences of GUI events that will be exercised by the end users / widgets. This is so because only by exploring the events in the GUI, the user of the system could accept or opt for changes. As an example, for a calculator application, the GUI could have been designed in such a way that the results of both the scientific operations and arithmetic operations are displayed in the same screen, but the customer might want the scientific operation in a separate window and the arithmetic operation in a separate one. Regression testing is an important software maintenance activity for traditional software, accounting for as much as one-third of the total cost of software production. However for GUI based software systems, regression testing still remains as an unexplored area. Regression testing research has focused on the development of regression test selection techniques that choose a set of test cases that represent correct input and are deemed necessary to validate the modified software from the existing test units. Regression testing in the area of GUI based software systems has proved that many test cases of the existing test suites in the GUI based software systems that undergoes frequent modifications are recognized as obsolete test cases (test cases that cannot be rerun).

In this work we propose a system that facilitates that automated test case generation which reduces the cost of rewriting the modified test cases. The system has three phases namely generation of test cases for the new/original system, construction of the Event Flow Graph that represents the existing test suite and the generation of test cases for the modified GUI system using the Depth First Search method.

II. Literature Survey

A. Test Tools

There are various tools available to perform testing. QTP (Quick Test Professional) is especially designed for testing the web based applications. WinRunner, Test-Complete facilitates regression, load and stress testing, but these tools work on the basis of capture and replay technique [11]. Most available test tools works on the basis of record and replay techniques [3]. These tools record the normal execution of the software and runs software for various test cases and generates reports for each test run as passed or failed indicating the execution as success or not. Only a few of them present the actual output which can be compared with the expected output. Testing is performed by setting the checkpoints [11]. To set the checkpoints and set the test cases for a field, a sequence of actions has to be performed in the GUI to reach the desired point or field. There are possibilities that a tester would leave an action unexplored which could be considered for a checkpoint. And therefore locating a component in a complex GUI would become a time consuming task [7].

B. Related Research Work of the System

A few automated GUI test case generation techniques have been proposed [2]. However, they all require creating a model of the GUI, a significant resource intensive step that intimidates many practitioners and prevents the application of the techniques. The GUI Ripping technique [8], [4] and [6] is to reverse engineer the GUI's model directly from the executing GUI. Once verified by the test designer, this model is then used to automatically generate test cases. GUI ripping has numerous other applications such as reverse engineering of GUI products to test them within the context of their use, porting and controlling legacy applications to new platforms, and developing model checking tools for GUIs [13]. The testing process in GUI ripping is a dynamic process that is applied to executing software's GUI. Starting from the software's first window (or set of windows), the GUI is "traversed" by opening all child windows. All the window's widgets (building blocks of the GUI, e.g., buttons, text-boxes), their properties (e.g., background-color, font), and values (e.g., red, Times New Roman, 18pt) are extracted. Developing this process has several challenges that are required to develop the solutions. For example, some windows may be available only after a valid password has been provided [8]. Since the GUI Ripper may not have access to the password, it may not be able to extract information from such windows. Another process and tool support is required to visually add parts to the extracted GUI model. GUI Ripper is used as a central part of two large software systems called GUITAR1 (GUI Testing Framework) and DART (Daily Automated Regression Tester) to generate, execute, verify GUI test cases, and perform regression testing [2]. The paper has provided details of two instances of the GUI Ripper, one for Microsoft Windows and the other for Java Swing applications. But, one main challenge is to provide rapid feedback to the developers about parts that may have been inadvertently broken during maintenance which cannot be captured using this technique.

Smoke tests have become widespread as many software developers/maintainers found them useful. Popular software that use daily/nightly builds includes WINE, Mozilla, AceDB, and open webmail. During nightly builds, a development version of the software is checked out from the source code repository tree, compiled, linked and smoke tested. Typically unit tests and sometimes acceptance tests are executed during smoke testing. Such tests are run to (re)validate the basic functionality of the system. The smoke tests exercise the system completely, such a way that they don't have to be an exhaustive test suite but they should be capable of *detecting major problems. A build that passes the smoke* test is considered to be a good build. Bugs are reported, usually in the form of emails to the developers. Frequent building and re-testing is also required because new software development processes advocate a tight development/testing cycle. A number of tools support daily builds, some of the popular tools include Cruise Control, IncrediBuild, Daily Build, and Visual Build.

A limitation of the currently available nightly builds is inadequate testing and re-testing of software that has a GUI. Frequent and efficient re-testing of conventional software requires automated regression testing [10], which is a software maintenance activity, done to ensure that modifications have not adversely affected the software's quality. Although there has been considerable success in developing techniques for regression testing of conventional software, regression testing of GUIs has been neglected. Consequently, there are no automated tools and efficient techniques for GUI regression testing. Not being able to adequately test GUIs has a negative impact on overall software quality moreover GUIs today constitute as much as 45-60% of the total software code [4]. Currently, three popular approaches are used to handle GUI software. First and most popular, is to perform no GUI smoke testing at all, which either leads to expensive or time consuming GUI testing later. Secondly, to use test harnesses that call methods of the underlying business logic as if initiated by a GUI. This approach not only requires major changes to the software architecture, it also does not perform testing of the end-user software. Thirdly to use existing tools to do limited number of GUI testing. Examples of some tools used for GUI testing include extensions of JUnit such as JFCUnit, Abbot, Pounder, and Jemmy Module2 and capture/replay tools such as WinRunner3 that provide very little automation, especially for creating smoke tests. Developers/maintainers who employ these tools typically come up with a small number of smoke tests.

In reusing the test case, the reusability of the test case has to be verified and a technique is required to make unusable test cases as reusable for the modified version of GUI. Techniques to check the reusability of the test cases and to make unusable test case reusable for the modified version, the technique used by Memon [5] is used. Those techniques are used for GUI developed with java, in the proposed paper we have utilized this technique to work for GUI designed with HTML.

III. Modeling of the System

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 'S0' of a test case is not one of the valid initial states 'SI' for the modified software, then it cannot be repaired because the valid initial state is required to identify repairing action to be taken. If 'S0' belongs to 'SI', 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. The comparisons of the two EFG's is as follows: If EFGo and EFGm 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 consideration.

(1) The set of all vertices deleted from the original EFG:

Vertices deleted ← Vertices (EFGo)-Vertices (EFGm);

(2) The set of edges deleted from the original EFG:

Edges deleted ← Edges (EFGo)-Edges (EFGm);

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 than one set of modifications, and a test case may be marked as unusable several times because of a single modification on an event here, 'e'. Being marked as unusable several times has no effect on the reparability of the test case. Once the unusable test cases have been identified, they are repaired by the test case repairer.

Approach for repairing test cases; the test-case repairing approach is based on user-defined transformations that deletes or inserts 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. 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, (1) 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: $< e (i-1); e (j) >$ 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 [5]. A variant of this technique is used in our work to reuse the test case.

A. Modeling of GUI

Model of GUIs that was developed for a GUI testing framework is presented in this section. A GUI is modeled as: a set of objects O = {o1, o2, …, om} (e.g., hyperlinks, forms, buttons) and a set of properties P = {p1, p2, …, pl} of those objects. Each GUI uses certain types of objects with associated properties, at any specific point in time. The state of the GUI is described in terms of all the objects that it contains, and the values of all their properties. Formally, the needed definitions for GUI are as follows:

1. Definition of state:

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.

A set of states 'SI' is called the valid initial state set for a particular GUI if the GUI may be in any state $Si \in SI$ when it is first invoked. The state of a GUI is not static; events performed on the GUI change its state and these states are called reachable states. The events are modeled as state transducers.

2. Definition of event:

The events E = {e1, e2, . . . , en} associated with a GUI are functions from one state to another state of the GUI. The function notation Sj = e (Si) is used to denote that 'Sj' is the state resulting from the execution of event 'e' in state 'Si'. Events occur as part of a sequence of events. Of importance to testers are sequences that are permitted by the structure of the GUI. Test case generation is restricted to such legal event sequences, defined as follows:

A legal event sequence of a GUI is e1; e2; e3; . . . ; en; where ei+1 can be performed immediately after 'ei'. 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), whereas <Open, Select, Cut> is legal.

B. Modeling of Test Cases

1. Definition of GUI test case:

GUI test case 'T' is a pair (S0, e1; e2; . . .; en), consisting of a state S0 ∈SI, called the initial state for 'T', and a legal event sequence "e1; e2; . . . ; en". 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.

GUI test case (S0, e1; e2; . . . ; en) is unusable if a modification of a GUI causes the state 'S0' to not be reachable in the GUI or if the sequence "e1; e2; . . . ; en" cannot execute to completion. Unusable test cases cannot be executed on the GUI and are usually discarded.

C. Modeling of Event Flow Graph (EFG)

Flow of events in the GUI is modeled as event flow graph. Components that have actions are identified as events. An event that triggers a page is denoted by "page-event" and here there are two pages associated, the "page1" where the event triggering a page is present and the "page2" which is the triggered page. All the pages are events but the converse is not always true. Events triggering the pages are named as "page1-page2" otherwise name of the event is extracted from the front end program. From "Fig. 1", we see the event "client" in page "loan.html" triggers the page "loan0.html", therefore the event "client" is stored as "loan.html-loan0.html". Similarly, the event "Apply for Loan" in page "loan0.html" triggers the page "loan3.html", therefore the event "Apply for Loan" is stored as "loan0.html- loan3.html". An event "Reset" is an event in "loan3.html" page, this event is stored as "loan3.html-Reset". Also, the event "Apply" present in page "loan3.html". Therefore within a page, when an event triggers to a different page, both the events and 'page-e' of that page are identified and stored as events. Before a page is processed it is necessary to check the page is visited earlier or not to avoid processing a page redundantly.

The events identified are the nodes or vertices of the EFG. If an event 'ej' is accessible after 'ei' is explored, then it means an edge exist between 'ei' and 'ej', its direction is from 'ei' to 'ej', which means 'ei' is the starting node and 'ej' is the terminating node. Edges can exist between 'ei' and 'ej' though 'ei' and 'ej' appear in different pages, in case 'ei' triggers the page in which 'ej' appears. Edges are represented as, pagei-eventi->pagei+1-eventi+1 where, eventi+1 is accessible after eventi is explored. pagei+1 and pagei represent the pages in which eventi+1 and eventi appears respectively.

In "Fig. 1", once the event "Apply for Loan" is explored i.e. loan.html-loan3.html is explored, the event "Apply", "Reset" and "Main Page" i.e. loan3.html-Apply, loan3.html-Reset, loan3.html-loan0.html are accessible. Thus the edges are represented as,

loan.html-loan3.html->loan3.html-Apply,
loan.html-loan3.html->loan3.html-Reset,
loan.html-loan3.html->loan3.html-loan0.html

respectively.

"Fig. 2" represents other edges that exist for the GUI represented in "Fig. 1"

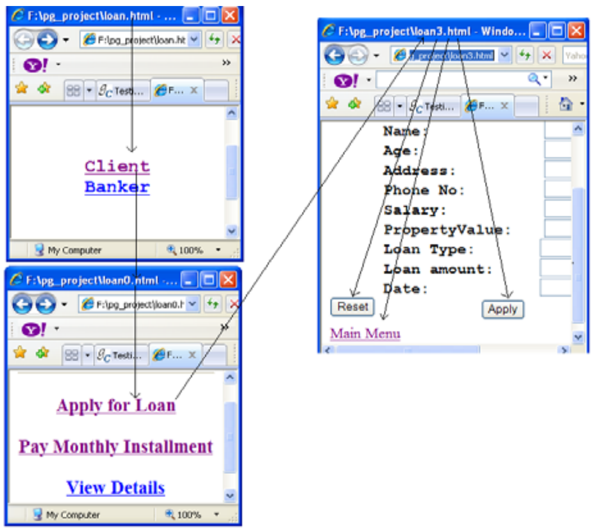
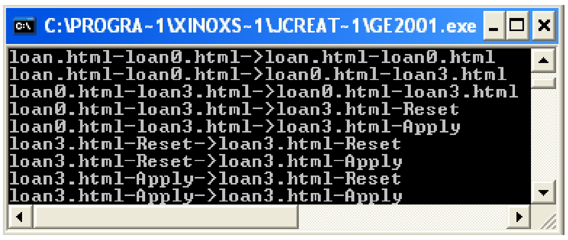


Figure 1. Page-events



```
C:\PROGRA~1\XINOXS~1\JCREAT~1\GE2001.exe
loan.html-loan0.html->loan.html-loan0.html
loan.html-loan0.html->loan0.html-loan3.html
loan0.html-loan3.html->loan0.html-loan3.html
loan0.html-loan3.html->loan3.html-Reset
loan0.html-loan3.html->loan3.html-Apply
loan3.html-Reset->loan3.html-Reset
loan3.html-Reset->loan3.html-Apply
loan3.html-Apply->loan3.html-Reset
loan3.html-Apply->loan3.html-Apply
```

Figure 2. Edge-events

Edges also exist between the events appearing in the same page. From "Fig. 1", event "Reset" in loan3.html is accessible after event "Apply" in the same page is explored. This is represented as, loan3.html-Apply->loan3.html-Reset. When all the events in the GUI are identified and all the possible edges between the events are identified, altogether they become the event flow graph.

Pseudo code for EFG Construction

```
Get the file name of the first page of the GUI from the
user
Extract the tags
Identify the components that has actions
Identify the events that triggers display of next page
If (page is visited)
{   Skip the page }
Otherwise Goto step 2
Store the event detail in a file along with the page to
which it belongs to
If (event triggers a new page)
{           For (each of the event in the triggered page) {
```

```
            stored edge in a file with triggering event as
            start node and triggered event as the
            terminating node
    } Constitute all possible edges into EFG
```

IV SYSTEM ARCHITECTURE

System Architecture is shown in "Fig. 3". To reuse the test cases for the modified version of GUI, the test cases generated for the original version of GUI, EFG of the original version of GUI and EFG of the modified version of GUI are required as input to the process. The final output from the system is set of test cases for the modified version of GUI in such a way that all the events in the modified GUI can be explored. The test suite for the modified version of GUI, which is actually output of the system, is not regenerated instead the test case generated for the GUI before modification is reused at the most. The final test suite of modified version of GUI is composed of the following:

(a) set of test cases directly reused from the test suite that is generated for the GUI before modification,

(b) set of modified test cases those which were actually test case for the GUI before modification and unusable test cases after modification in the GUI, and made usable for testing modified version of GUI after treating the test case with the transformations, and

(c) set of test cases newly generated for those events which were not covered by either the directly reused test case or the modified test cases.

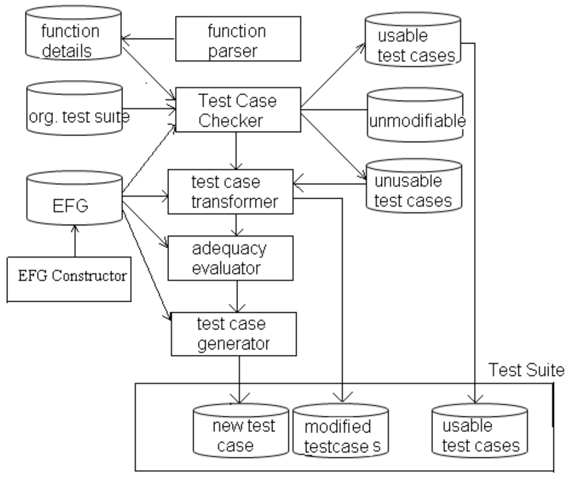


Figure 3. System Architecture

Test suite used to test the GUI before modification is maintained in repository as original test suite. EFG is generated from the model for EFG constructor as explained in Section III-C, EFG constructor parses through the HTML program, reads the tags and identifies the components and lists the components with action as events. Identifies the events that triggers a page, then parses through that page and identifies the events present in that page, EFG performs this action recursively until all the pages and events in the GUI are identified. The EFG constructor maintains the details about the events to which page an event belong to and which event triggers a page. With these information edges are identified and the complete EFG is maintained in a file. This EFG is used to identify the difference between the GUI before modification and after modification. EFG is also used to identify the solution to treat an unusable test case and to check the test case adequacy.

Test case checker partitions the original test suite into three parts as (i) usable test cases, (ii) unusable test cases and (iii) un-modifiable test cases. (i) Usable test cases are those which can be directly reused by the modified version of GUI, test case checker parse through the test cases one at a time and identifies the events a test case covers, it then checks the existence of those events with the EFG of modified GUI, if all the events covered by a test case also exist in the EFG of modified GUI, then that test case is added to set of usable test cases. Similarly, (ii) unusable test cases are those which cannot be directly reused by the modified version of GUI but can be made usable by treating it, test case checker parse through the test case one at a time and identifies the events a test case covers, and checks the existence of those events with the EFG of modified GUI, if an events covered by a test case does not exist in the EFG of modified GUI, then that test case is added to the set of unusable test cases. (iii) Un-modifiable test cases are those which cannot be reused by the modified version of GUI, test case checker parse through the test case one at a time and identifies the events a test case covers, and checks the existence of those events with the EFG of modified GUI, if the starting event of a test case does not exist in the EFG of modified GUI, then that test case is added to set of un-modifiable test cases. List of events deleted and list of edges deleted reused by the Test case checker to perform the above mentioned task.

Test case transformer transforms the unusable test case into usable test case for the modified version of the GUI as per the transformations mentioned in Section VI-C. Test case transformer identifies the problem point in the unusable and partitions the test case into two parts as correction point and resume point. The point in the test case that made the test case unusable i.e. the location in the event sequence where a deleted event occurs is the problem point. The correction point is the test case fragment from starting point to event before the problem point of the test case and resume point is the fragment from an event next to problem point to the end point of the test case. With the help of EFG constructed for the modified version GUI, Test case transformer identifies the bridge between the correction point and the resume point such that by appending correction point, bridge and the resume point, the Test case transformer arrive with a modified test case that is usable for the modified version of GUI. The bridge between correction point and resume point is identified few types of transformations. They are explained in Section VI-C of this paper.

Adequacy evaluator checks the test case adequacy by verifying the event coverage. Adequacy evaluator takes reused test cases, modified test cases and EFG as input. Events covered by all the reused test cases and modified test cases are extracted and cross checked with the events in the EFG of the modified version of GUI. Events that are present in the EFG of the modified version of GUI and absent in the set of events covered by all the reused test cases and modified test cases are taken as uncovered events. These uncovered events are fed as input to test case generator.

Test case generator generates test case for the uncovered events. Test case generator identifies the event with in-count 0 as starting events from the EFG. Test case generator takes starting event as correction point and uncovered event as resume point and searches for the bridge between correction point and resume point. By appending starting event, bridging event sequence and uncovered event, test case generator arrives with new test case for uncovered event.

V. Test Case Generation

In testing GUI, test cases are sequence of events. Events can be sequenced with two approaches namely; breadth first approach and depth first approach [12]. Depth first approach is undertaken, as that is found suitable for traversing the GUI. For each node in the event flow graph i.e. for each event, the number of edges leading into a graph: the in-degree and leading out of a graph: the out-degree is found. Set of initial events and final events are identified. Events with in-degree=0 are identified as starting events and events with out-degree=0 are identified as terminating events. Events are sequenced by traversing the EFG. The representation of the events, their in-degree followed by out-degree of each of the event in the GUI are for example, “loan.html-loan0.html, 0, 1”; “loan0.html-loan3.html, 1, 2”; loan3.html-Reset, 1, 0”; “loan3.html-Apply, 1, 0”; etc., Every event sequence starts from initial event and ends with terminating event as shown as “loan.html-loan0.html->loan0.html-loan3.html->loan3.html->loan3.html-Reset”, “loan.html-loan0.html->loan0.html-loan3.html->loan3.html->loan3.html-Apply”, etc, From the event sequence test cases are generated in such a way that all the events in the GUI are explored.

Pseudo code for Test Case Generation

```
Parse the EFG
For ( nodes )
{   Find indegree
    Find outdegree
    Store as an object
} If ( indegree==0 )
{   Enqueue to starting node }
For ( starting nodes )
{       Dequeue node
        For (nodes)
        {       If (node is adjacent node of dequeued node)
        {Indegree[node] - -
                }       If (indegree==0)
                {Enqeue node
                }Do steps for starting nodes.
        }
} Write test cases to a file
```

VI. Reusing GUI Test cases

Reusing GUI test cases prevents tester from regenerating the test case completely. By reusing the test cases time consumed to regenerate the test case by capture replay technique can be reduced. Automated reusing of test case is beneficial rather than regenerating the test case in an automated manner, because by reusing the test case, we could separate those test cases that actually examine the modified event or edge. Those test cases that are modified test cases and newly generated test cases cover modified events and uncovered events respectively. Thus modified test cases could be given higher priority than the other test cases so that the modified events are examined.

The test case reuse technique used in this paper takes into account the changes made in navigation among the pages but do not consider the structural changes. As structural changes do not affect the event sequence and the test cases remain unaffected for structural changes. Thus the technique is limited to changes in navigation among the GUI pages.

To reduce the number of test cases it is necessary to identify the changes made to the original GUI. EFG of GUI generated listing the generation of test cases is maintained for later use while reusing the test case. For the modified GUI, EFG is generated in the same way as generated for the original GUI.

A. Identify Usable and Unusable Test Cases

From the EFGs of original and modified GUI, all the events (nodes) and edges are retrieved and existence of each event in the original GUI are cross checked with the events in the modified GUI. If an event in original GUI is absent in the modified GUI, that event is added to the list of deleted events. In the same way, existence of each event in the modified GUI is cross checked with the events in the original GUI. If an event in modified GUI is absent in the original GUI, that event is added to the list of events added. Similarly existence of each edge in the original GUI is cross checked with the edges in the modified GUI. If an edge from the original GUI is absent in the modified GUI, that edge is added to the list of deleted edges. In the same way existence of each edge in the modified GUI is cross checked with the edges in the original GUI. If an edge in modified GUI is absent in the original GUI, that edge is added to the list of added edges.

Reusability of the test cases is performed to identify those test cases that are directly usable and those test cases that has to be repaired to make them reusable for the modified version of GUI. The test cases, that have at least one event that is deleted from original GUI becomes unusable for modified GUI. To check the reusability of the test cases, each is retrieved one at a time and the events it covers are taken as a list and the existence of event in that list is verified with the list of events deleted. If a test case has an event that has been deleted, then that test case is added to unusable test case list. Each of the test cases generated for the original GUI is verified in the same way as explained above.

B. Modifying Unusable Test Cases

Once the reusability of the test case is determined and is arrived with a list of usable and unusable test cases, the unusable test case has to be repaired or modified to make them usable for the modified version of GUI. Unusable test cases are retrieved one at a time and the repairing technique explained in Section V of this paper has to be performed. Each unusable test case will have a problem point. Problem point is the point in the test case, that made the test case unusable that is the location in the event sequence where a deleted event occurs. Then the test cases are segmented into two parts, as correction point and resume point. The test case fragment from starting point to event before problem point is taken as the correction point and the fragment from an event next to problem point to end point is taken as the resume point

Now a bridge has to be identified between connection points and resume point. Bridge could be an event or event sequence that acts as the path to reach resume point from the connection point. Thus by appending bridge to connection point and then by appending the resume we could arrive with a modified/repaired rest case whose event sequence is valid in the modified GUI.

At this point it should be noted that a test case may not have only one problem, a test case could have more than one puncher or problem point. Our technique as explained above segments test case into two fragments as correction point and resume point. This segmentation is done based on location of problem point that appears first, but it is true that the test case could have other problem points also. So, the modified test case has to be rechecked to determine its reusability. If the modified test case is usable i.e., it does not have any problem point then it is added to modified reusable test case list. Otherwise, the test case has to be modified to repair the other problem points. By performing this process recursively, we could arrive with a final modified test case that has no problem point at all.

C. Types of Transformations

Transformation is the process of converting an unusable test case into reusable / modified test case. As explained before, to make unusable test case reusable for modified version of GUI, a bridge has to be identified between correction point and resume point. To say exactly connecting event or event sequence is required to bridge between end node of correction point fragment and starting node of resume point fragment.

SNORP is the starting node of resume point. ENOCP is the end node of correction point. ENE is the event of end node which is derived from ENOCP. ENP is the page of end node which is derived from ENOCP. SNP is the page of starting node that is derived from SNORP. And the various types of transformations that are used in different scenarios are explained as follows:

Case (i) ENOCP and SNORP occur in the same page.

Case (i) has the following two possibilities

(i)a. ENE does not triggers a page

(i)b. ENE triggers a page

For case (i)a. Transformation-1 is used. When SNP and ENP are same and ENE does not trigger a page, it means that, once the event ENOCP is explored, the state remains in the same page where SNORP exists, thus the SNORP can be explored immediately after ENOCP. Thus the repairing / transformation task is to check the existence of edge from ENOCP to SNORP. This edge will be the bridge to reconstruct the unusable test case so that it becomes reusable after modification made to it. The modified test case is derived by appending correction point, bridge and resume point as represented below,

("->" is the bridge in this transformation)

correction point + "->" + resume point

--Transformation-1

For case (i)b. Transformation-2 is used. When SNP and ENP are same and ENE triggers a page, it means that, once the event ENOCP is explored, the state of current page displayed will be changed, that is the user will be taken to some other page. Thus a bridging event has to be identified with page name same as ENE and event name same as SNP. When bridging event does not exists, and then an event sequence that could be used as a bridge between ENOCP and SNORP has to be identified. While identifying the vent sequence, it should be noted that the event sequence are examined with depth first search algorithm. The bridging event sequence must have first node with page name same as ENE and end node event same as SNP. The modified test case is derived by appending correction point, bridge and resume point as represented below,

correction point + "->" + bridging event + "->" + resume point

correction point + "->" + bridging event sequence + "->" + resume point --Transformation-2

Here bridging event or event sequence acts as the bridge to repair the test case.

Case (ii)ENOCP and SNORP occur in different pages.

ENOCP and SNORP occur in the different page, i.e. ENP is different as SNP. For the above said condition, there are two possibilities as follows,

(ii)a. ENE triggers SNP

(ii)b. ENE does not triggers SNP

For the case (ii)a. the Transformation-3 is used. When SNP and ENP are different and ENE triggers a page, it means that, once the event ENOCP is explored, the state will be taken to a page where SNORP exists, thus the SNORP can be explored immediately after ENOCP. Thus the repairing / transformation task is to check the existence of edge from ENOCP to SNORP. This edge will be the bridge to reconstruct the unusable test case so that it becomes reusable after modification made to it. The modified test case is derived by appending correction point, bridge and resume point as represented below,

("->" is the bridge in this transformation)

Correction point + "->" + resume point

--Transformation-3

For the case (ii)b. Transformation-4 is used. When SNP and ENP are different and ENE does not triggers SNP, it means that, once the event ENOCP is explored, the state of current page displayed will be changed, that is the user will be taken to some other page. Thus a bridging event has to be identified with page name same as ENE and event name same as SNP. When bridging event does not exists, and then an event sequence that could be used as a bridge between ENOCP and SNORP has to be identified. While identifying the vent sequence, it should be noted that the event sequence are examined with depth first search algorithm. The bridging event sequence must have first node with page name same as ENE and end node event same as SNP. The modified test case is derived by appending correction point, bridge and resume point as represented below,

correction point + "->" + bridging event + "->" + resume point

correction point + "->" + bridging event sequence + "->" + resume point

--Transformation-4

Here bridging event or event sequence acts as the bridge to repair the test case.

Case (iii) Correction Point is null

In this case "correction point is null" implies that an initial event or initial sequence of events are deleted. Procedure for the transformation is as follows. Set of starting nodes of the GUI are identified by examining the in-count of each of the events/ nodes in the EFG. Events /nodes with in-count 0 are identified as starting nodes. Then the event or event sequence has to be identified in such a way that, the event or event sequence could act as a bridge between the starting node and the SNORP.

--Transformation-5

Case (iv) Resume Point is null

In this case "resume point is null" implies that terminating event or terminating sequence of events are deleted, the transformation is done by eliminating the resume point and take correction point as the test case.

--Transformation-6

VII. Performance Evaluation

Performance of the system is evaluated with applications whose GUI are designed with HTML code that are downloaded from the open source sites. Test cases are generated for those applications manually and the time taken is observed. The system developed is executed with the same applications as input to generate test case for them automatically. Interesting measures/attributes identified to measure the performance of test case generation in our work are: number of events, number of edges, number of pages, number of test cases generated, time taken to generate test cases automatically, time taken to generate test cases using capture/replay tool.

All these attributes are measured and recorded. Number of events is always less than the number of components, as those components that do not have action (example-labels) are not considered as events. It is noticed from the observation that test cases are generated in seconds, when generated automatically with the system, whereas, if test cases are generated manually it takes minutes for small applications and hours for large applications. Thus with system been developed test cases are generated in seconds for GUI designed with HTML and thus reduces the time spent in generating the test case and the effort spent for the same.

Attributes affecting reusing test cases identified in our work are: number of original events, number of modified events, number of original edges, number of modified edges, number of original test cases, number of test cases usable, number of test cases unusable, number of test cases usable after modified, percentage of GUI modification and percentage of test cases reused.

All these attributes are measured and recorded. It is observed that for a small structural change in the GUI most of the test cases become unusable. Percentage of test cases that became unusable differs for each application depending on the number of events or edges modified and in which page they are present. Those test cases that became unusable are treated with this system to make them usable for the modified version of GUI. The usability of the test case after treating the test case that has actually become unusable for modified version of GUI are presented as a graph in Fig 4.

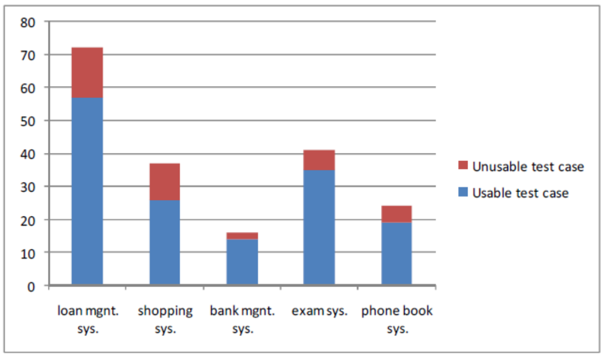


Figure 4. Graph Plot showing usable and unusable test cases after modification.

The blue portion of the graph represents those test cases that were actually unusable for the modified version of GUI and m ade usable after treating it. The red portion of the above graph represents those test cases that are unusable for the modified version of GUI even after treating it.

Percentage of Test Case Reused = (number of test cases unusable / number of test cases usable after modified) *100 (1)

On an average, almost 80% of unusable test cases are made usable after treating them with the system been developed. For uncovered events test cases are generated automatically without any human intervention, which actually reduces time spent.

VIII. Conclusion and Future Work

In this work, we developed a system that focuses on test case generation and reusing testing cases in the event of GUI modification. In our system we assume that GUI is designed with HTML. EFG is constructed for GUI by traversing the pages in depth first search technique and then the test cases are generated as sequence of events by traversing through the EFG. The test cases are generated automatically in seconds without any human intervention.

The system also addresses reusing test cases for GUI. EFGs are constructed by parsing the html program of the original and modified GUIs. EFG is constructed by identifying the components that has action and by identifying links between the events in the GUI. Constructed event flow graph is a directed graph indicating triggering event in the start terminal and triggered event as the end terminal of an edge in the graph. It is also un-weighted graph. EFG is stored as a set of edges in a file.

Test cases of original GUI are validated with the Modified EFG and distinguished as usable test case and unusable test case. For a small change in GUI most test cases becomes unusable. Those unusable test cases are transformed in such a way that with few changes in unusable test case they become usable for the modified GUI. Had we gone for Regenerating test cases for slightly modified GUI it would have become very expensive.

The technique used to reuse the test case has high chance of making few test cases redundant. A test case would have been modified to make them usable but the events explored by that particular event might be included in one test case or the other, in that case the test case becomes redundant. Research can be continued in this area to eliminate redundant test cases and reduce number of test cases with the criterion to cover all the events in the modified GUI.

References

[1] E. O. Ariss , Dianxiang Xu, S. Dandey, B. Vender, P. McClean, B. Slator, editors. A systematic capture and replay strategy for testing complex GUI based Java applications. TNG '10 Proceedings of the Seventh International Conference on Information Technology: New Generations; 2010; Washington, DC, USA: IEEE Computer Society; 2010. 1038–1043.

[2] Paul Gerrard. Testing GUI Applications [internet]. Edinburgh, UK: Eurostar Conference; Nov 1997 [updated 2010; cited 2010 Dec 11]. Available from: http: //gerrardconsulting.com/?q=node/514.

[3] Rick Hower. Software QA and Testing [Internet]. [place unknown]: 1996 Nov 22 [updated 2011 Jan 3; cited 2011 Jan 9]. Available from: http: //www.softwareqatest.com/qatfaq1.html

[4] A. M. Memon, GUI testing: pitfalls and process, IEEE J Computer. 2002 Aug; vol.35 (8): 87-88.

[5] A. M. Memon. Automatically repairing event sequence-based GUI test suites for regression testing. ACM Transactions on Software Engineering and Methodology (TOSEM). 2008 Nov; 18 (2).

[6] A. M. Memon, A. Nagarajan, Qing Xie. Automating regression testing for evolving GUI software. Journal of Software Maintenance: Research and Practice – Special issue: 2003 International conference on software maintenance: The architectural evolution of systems. Jan; 17 (1); 27–64.

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

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

[9] A. Murgia, R. Wolff, W. Steptoe, P. Sharkey, D. Roberts, E. Guimaraes, A. Steed, J. Rae, editors. A tool for replay and analysis of gaze-enhanced multiparty sessions captured in immersive collaborative environments. 12th IEEE/ACM International Symposium Distributed Simulation and Real-Time Applications; 2008; Washington, DC. USA: IEEE Computer Society; 2008. 252–258p.

[10] R. M. Poston. Automating specification-based software testing. 1st. ed. IEEE Computer Society, CA, USA: IEEE Computer Society Press; 1997. 272 p.

[11] Vyom Network [Internet]. [place unknown]: 2003 [cited 2010 Aug 23]. Available from: http: //www.onestoptesting.com/.

[12] M. A. Weiss. Data structures and algorithm analysis in C. 2nd. ed. Boston, USA: Addison-Wesley Longman Publishing Co., Inc; 1997. 511 p.

[13] Qing Xie, Memon A M, editors. Model-Based testing of community-driven open-source GUI Applications. Proceeding ICSM '06 Proceedings of the 22nd IEEE International Conference on Software Maintenance; 2006; Washington DC, USA; c2006. 145 p.