ISSN:2231-0711

Bouchaib Falah et al | IJCSET |March 2013 | Vol 3, Issue 3, 76-79

# GUI Regression Test Selection Based on Event Interaction Graph Strategy

Bouchaib Falah,Rahima Nouasse,Yassine Laghlid

*School of Science and Engineering*
*Al Akhawayn University*
*Ifrane, Morocco*

***Abstract*-- Testing is an efficient mean for assuring the quality of software. Nowadays, Graphical User Interfaces (GUIs) make up a big part of applications being developed. Within the scope of regression testing, some test cases from the original GUI are usable and others are unusable. This paper presents an algorithm that drops the unusable test cases and creates new test cases based on the main differences between the two GUIs, which are represented as uncovered edges. Furthermore, the algorithm creates a new test suite for the modified version by combining the usable test cases and the new created test cases.**

***Keywords*— regression testing; reusable test cases; event interaction Graph; flow**

## I. INTRODUCTION

Graphical User Interfaces (GUIs) are common in today's most applications, ranging from networking systems and operating software to desktop applications. In fact, a GUI is an interface for users to send commands to and monitor the underlying business logic behind the application. Indeed, the quality of GUIs directly and basically influences the reliability and usability of the entire software application. A faulty GUI affects the quality of a software product, reducing user satisfaction. In this context, the quality assurance of the GUIs is indispensable. Black box testing of applications with Graphical User Interfaces (GUIs) can be accomplished by running sequences of events based on a model of the Graphical User Interfaces. Test cases identify sequences of behavior; which means, these are subsets of the specifications of behavior for the application. Several Researches have proved that testing a GUI from this standpoint will find errors associated not only to the Graphical User Interface and its source code, but to the underlying business logic of the software system as well [10].

A modern methodology developed to acquire such a model is to perform the application under test and analyze or rip the handled events and connections between them. These events can then be represented as a graph showing their flow (Event Flow Graph) or more conceptually as a directed graph of events interacting (Event Interaction Graph) [7]. These graphs will be explained later in the background section. These models are then used to merge and test sequences of events for program validation. These models are effective at producing short sequences of events for generating test cases; these test cases have a length of 2 or a very small number. Such sequences can be run automatically and quickly by means of a test harness. If a Graphical User Interface has 6 events, where every event can be run after every other event, then there are only 6^2 or 36 length 2 sequences and only 216 length 3 sequences [7].

Previous conducted studies have proved that longer event sequences can detect errors that are generally missed by short ones. These longer sequences get to more compound code in the application. On the other hand, there are two primary restrictions of testing Graphical User Interfaces by means of long test sequences. First, the number of sequences increases exponentially with length. In the 6 event Graphical GUI, if we generate length 10 sequences, we have a possible pool of 6^10 single sequences [9]. Moreover, this procedure suffers from the second problem which is the infeasibility of test cases. Based on this example, we can notice that GUI regression testing is very expensive and it projects an exponential growth. When a specific graphical user interface is altered, it is costly to construct new test suites for it and ignore the old test suites which are created for the original GUI.

Therefore, we propose a technique for repairing the Graphical User Interfaces test suites by searching for new feasible test cases to complete the coverage of feasible combinations and adding them to the set of usable test cases generated in the original GUI. This research paper makes the following contributions: 1) Presenting an intuitive algorithm for repairing GUI test suites. This algorithm takes a GUI and its modified version, detects the infeasible along with the unusable, and based on an intelligent aspect it creates new test cases and add them to the test suite of the modified version. 2) Conducting experiments on 2 simple examples to investigate the feasibility of the technique.

## II. RELATED WORK

In more modern work (Si Huang) [9], a novel feedback based method for Regression Graphical User Interface testing was proposed. This method necessitates an initial test suite to be created and run on the program under test [10]. Feedback from this execution is used to build a model of the Graphical User Interface and automatically produce additional test cases. In fact, the test suite is produced using the Event Interaction Graph (EIG) model. The suite is run on the GUI by means of an automatic test case tool. During test execution, the state of GUI is composed and exploited to automatically build an Event Semantic Interaction (ESI) projecting connections between events [9]. This relationship

demonstrates how a Graphical User Interface event is linked to another in terms of how it alters the other's running activities. The Event Semantic Interaction associations are used to create a new graph of the Graphical User Interface, named the Event Semantic Interaction Graph (ESIG). Since the test suite is created from the Event Interaction and the Event Semantic Interaction relationship, the Event Semantic Interaction Graph conveys specific features of the GUI. The Event Semantic Interaction Graph is used to create new test suites. These test suites have an indispensable property; each event is Event Semantic Interaction-associated to its succeeding event, which means, it was proven to impact the succeeding event during the running of the test suite [9].

Memon [4] has conducted regression testing studies and has proved that test cases can be repaired. When the organization of a GUI is altered, test cases from the original Graphical User Interface suite are either unusable or reusable on the modified Graphical User Interface [4]. Some algorithms were proposed to (a) automatically decide the usable and unusable test cases from a test suite after a Graphical User Interface modification, (b) find out the unusable test cases that can be fixed so that they can run on the modified GUI, and (c) use fixing transformations techniques to repair the test cases. The difficulties of fixing sequences were fewer in the context of regression testing because they exploited the differences between the old and the new Graphical User Interface [4].

Kepple [3] considered the issue of dipping the number of regression test cases to be executed. Their technique is to inspect the various modifications made to numerous parts of an application. If the modifications are within source code that is actually being run by a specific test in a regression test suite, then that specific test should be re-executed. Otherwise, it may be ignored, as that would lead to a very important conclusion which is a safe state without new code being added.

White Howard [4] used a methodology which includes the use of test suite capture data from a capture/replay testing tool. Based on the produced data, White could characterize a test suite for a provided Graphical User Interface using a call graph. This later is mainly based on scores that represent frequent paths selected in diverse test cases which are principally the critical paths. Therefore, these critical paths become indispensable for selecting which unit tests to execute.

## III. BACKGROUND: EVENT FLOW GRAPH (EFG)/ EVENT INTERACTION GRAPH (EIG):

An Event Flow Graph (EFG) conveys all potential event sequences that may be run on a Graphical User Interface [9]. It is a directed graph with nodes and edges that characterize a connection between events. An edge from node a to another node b means that the event represented by b may be executed straight away after the event represented by a of course along some implementation path. This association is named follows, which means, event b follows event a. The Event Flow Graph is modeled by a group of nodes N indicating events in the Graphical User Interface and a group E of ordered pairs (a, b), where {a, b} belongs to N, representing the directed edges in the Event Flow Graph; (a, v) belongs to E if b follows a [9].

Figure 1(a) illustrates a Graphical User Interface that consists of 4 events, New, Save, SaveAll, and File. Figure 1(b) shows the GUI's Event Flow Graph; the four nodes stand for the four events; the edges correspond to the follows relationships. In this Event Flow Graph, the event Save All follows File [9].

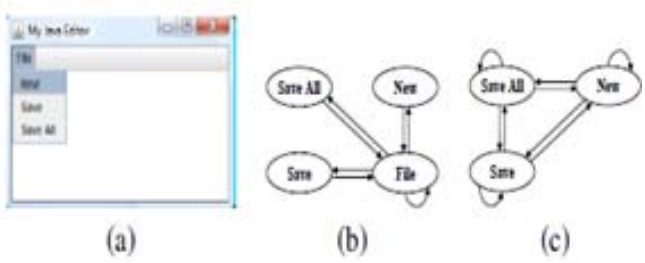


Figure 1. (a) A Simple GUI, (b) Its EFG, (c) Its EIG

Event interaction graph (EIG) nodes, on the other hand, do not symbolize events to open or close applications menus or windows menus. The output is a more solid, and thus more efficient, Graphical User Interface model. An Event Flow Graph can be automatically translated into an Event Interaction Graph by using graph rewriting conventions. Figure 1(c) illustrates the corresponding Event Interaction Graph. We can see that the Event Interaction Graph does not include the menu opening File event. The model used to obtain this Event Interaction Graph was to (a) remove File because it is a menu open event, (b) for all residual events a put back each edge (a,File) with edge (a, b) for each incident of edge (File, b), and (c) for all b, remove all edges (File, b). The Graphical User Interface's Event Interaction Graph is fully linked with 3 nodes representing the 3 events [9].

## IV. INNOVATIVE APPROACH

Let's define some terminologies that will be used in our approach.

**Definition1**: A node in a graph represents the interaction of a user with the system. The class node is composed of outgoing edges and incoming edges in addition to a Boolean variable Visited to make sure the node is not visited twice.

**Definition2:** an Edge connects two nodes. The first one is the source node and the second is the target node in addition to the label of the edge representing the relationship between the source node and the target node.

**Definition3:** a graph is basically a set of nodes with the connecting nodes.

Our approach consists of implementing a reusability algorithm that minimizes the cost and the time spent to generate new test cases for the modified version in regression testing. The test case reusability algorithm consists of comparing two event interaction graphs and come up with the

reusable test cases that can be rerun for the modified version in regression testing.

If the algorithm successfully matches every node and the set of outgoing edges of each node specified by a test case on the new model, then the corresponding test case is deemed reusable; otherwise, it is deemed as unfeasible.

The algorithm also labels all the edges covered by reusable test cases. After that, it builds a sub-graph containing all the skipped edges. It will also add outgoing edges of a choice node if any of them have been skipped. Finally, it generates new test cases from the sub-graph to achieve edge coverage.

This algorithm is a technique that not only identifies reusable test cases and generates new test cases but also selects test cases from a test suite.

## V. ALGORITHMS (PSEUDO-CODE)

The algorithm operates on two events interaction graphs as follows:

- The outgoing edges of the choice node in test case graphs should match all outgoing edges of the choice node in the new model graph.
- Check whether the number of outgoing edges for the current node agrees in the two graphs.
- If so, perform a pair-wise match of all outgoing edges in the two graphs.
- If any outgoing edge fails to match, the test case is identified as unfeasible. Note that if the label of a node is modified then the algorithm considers it as o modified. For example in figure 9 a node called file in paint has been changed to home in figure 10 even if the node is basically the same the test case is identified as unfeasible.
- When all the outgoing edges of a choice node are matched successfully, the algorithm continues to match the respective target nodes of all the outgoing edges
- When all the respective target nodes and their descendants in the test case graph match the new model graph, the whole test case is identified as reusable. From figure 6 and 7, we can see that the test case sequence <File, open, search, ok> and test case sequence <file, save> are the same in the modified version as well as in the original one. Thus those test cases will be reusable.
- Whenever two edges match successfully, they are added to the set of covered Edges.
- The algorithm then continues to recursively match the target node of the outgoing edge in the model graph with the target node of the outgoing edge in the test case graph.
- If two target nodes and their descendants match recursively, the option node is tagged as a match.
- However, if the two target nodes or any of their descendants fail to match, the algorithm will continue to try and match the outgoing edge of the node in the test case graph with other outgoing edges of the node in the new model graph. Taking the case of Paint, in figure 6 and 7, the algorithm didn't identify a match. In this case all the test cases will be generated for the modified version.

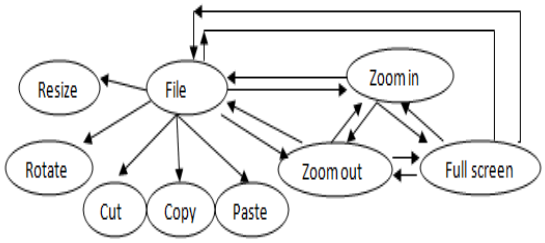


Figure 6. Original Event Interaction Graph for Paint

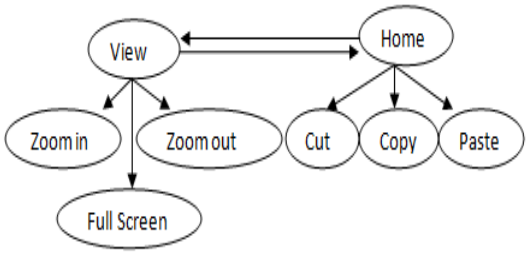


Figure 7. Modified Event Interaction Graph for Paint

- Any falsely remembered edges during the trial are removed from the covered Edges set. Finally, if the algorithm cannot match any of the outgoing edges of the model graph, the node is marked as a non-match, which means that the test case is unfeasible. Figure 8 shows unfeasible test cases which are identified in the edges: <File, Cut>, <File, Copy>, <File, Paste > , <Cut, Copy>, <Cut, Paste>, <Paste, Copy>

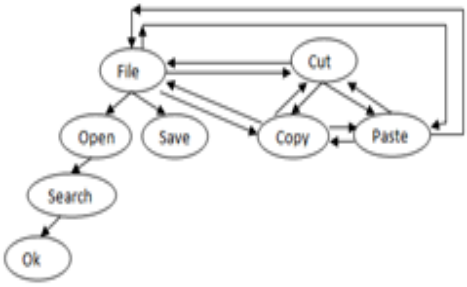


Figure 8. Original Event Interaction Graph for Word Pad

### A. *New Test Cases Generation*

Using the test case reusability algorithm, we have partitioned the test suite for the original model into reusable and unfeasible test cases. In addition, we have logged all the edges covered by reusable test cases. As we want to achieve edge coverage, we need a test case augmentation algorithm, which generates new test cases to cover all the skipped edges.

- It starts by finding all the skipped edges of the new model program graph based on all the edges covered by reusable test cases. From figure 9 defines the skipped edges : <edit,cut>, <edit,copy>, <edit,paste>, <edit,file>.

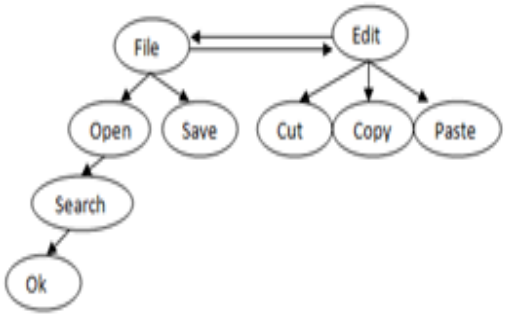


Figure 9. Modified Event Interaction Graph for Word Pad

- To cover the skipped edges with new test cases effectively, we first build a shortest path from the initial node of the model graph to the source node of each skipped edge. After that, we combine all the shortest paths to form a subgraph. In our example, the shortest paths are: Edit->cut, Edit->copy, Edit->paste, Edit>File
- We also add each skipped edge to the subgraph as shown in figure 10. Finally, we split the subgraph into new test case graphs in test normal form and generate the test cases from the new test case graphs to achieve edge coverage.

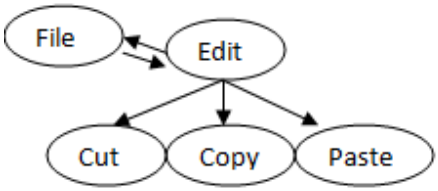


Figure 10. Sub-graph form shortest path and skipped edges

Our algorithm merges the original test case graphs of newly generated test cases and reusable test cases to form a test case graph for the modified model.

*B. Results Analysis*

Given a test case graph with m edges and the model program graph with n edges, in the worst case, each edge of the test case is compared with every edge of the model program graph, and so the complexity of the test case augmentation algorithm is O(nm). It should be emphasized that although there may be loops within a test case graph, our technique will not revisit the same node in the test case graph. As a result, the complexity of our algorithm is proportional to the size of the test case graph.

## VI. CONCLUSION AND FUTURE WORK

This paper presents a new regression testing algorithm that consist of test case reusability concept that minimizes the cost and the time spent to generate new test cases for the modified version in regression testing. In this work, we first explain the causes that may lead to infeasibility of test cases. We then present an intuitive algorithm that takes a GUI and its modified version, detects the infeasible along with the unusable, and based on an intelligent aspect it creates new test cases and add them to the test suite of the modified version. Finally, we experiment with this algorithm on a set of one simple example in order to check if this algorithm is feasible or not.

A possible future work will be to use these findings that we found in our study and continue the process to repair the found unfeasible test cases. Another possible future direction is to create a framework in which this proposed algorithm is integrated. This later will automatically detects unfeasible test cases and it will generate new test suite for the modified version of software.

## VII. REFERENCES

[1] Ball T. On the limit of control flow analysis for regression test selection. Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 1998), ACM SIGSOFT Software Engineering Notes 1998; 23(2):134–142.

[2] BEIZER, B. 1990. Software Testing Techniques, 2nd ed. Van Nostrand Reinhold, New York, NY.

[3] KEPPLE, L. R. 1994. The black art of GUI testing. Dr. Dobb's J. Softw. Tools 19, 2 (Feb.), 40.

[4] 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.

[5] MEMON, A., BANERJEE, I., AND NAGARAJAN, A. 2003. GUI ripping: Reverse engineering of graphical user interfaces for testing. In WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering. IEEE Computer Society Press, Los Alamitos, CA, 260–269.

[6] MEMON, A., NAGARAJAN, A., AND XIE, Q. 2005. Automating regression testing for evolving GUI software. J. Softw. Maint. Evolut. Res. Pract. 17, 1, 27–64.

[7] MEMON, A. M. 2001. A comprehensive framework for testing graphical user interfaces. Ph.D. dissertation, Department of Computer Science, University of Pittsburgh, Pittsburgh, PA.

[8] Memon AM, Soffa ML. Regression testing of GUIs. In Proceedings of the Joint 9th European Software Engineering Conference and 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC 2003/FSE-11), ACM SIGSOFT Software Engineering Notes 2008; 28(5):1-36.

[9] X. Yuan, M. B. Cohen, and A. M. Memon, "GUI interactiontesting: Incorporating event context," IEEE Transactions on Software Engineering, 2010, to appear.