Journal of Engineering and Applied Sciences 8 (5): 172-176, 2013
ISSN: 1816-949X
© Medwell Journals, 2013

Enhanced Approaches to Improve Graphical User Interface Testing Process

[1]E. Vijayakumar and [2]M. Punithavalli
[1]School of Computer Applications, Professional Institutions,
[2]Sri Ramakrishna Engineering College, Coimbatore, Tamil Nadu, India

Abstract: Graphical User Interfaces (GUI) are important components of Event-Driven software that are used mainly for improving user-computer interactions. As the number of graphical controls that the user can select using mouse or key board is very high, the number of test cases generated is also very high. Thus, the test cases generation process has to be optimized. This research performs this in three steps by enhancing the three operations, namely; test case generation, reduction and prioritization. Experimental results prove that the methods proposed have optimized the process of test case generation and has improved the accuracy of error detection rate. A maximum of 99.25% fault detection rate was obtained which shows that the proposed amalgamation of techniques are successful and can be used by the 21st century software.

Key words: Graphical user interface, test case generation, test case reduction, prioritization, classification

INTRODUCTION

Event-Driven Software (EDS) have rapidly become a critical part of business in many organizations. All EDSs take sequences of events (e.g., messages and mouse-clicks) as input, change their state and produce an output (e.g., events, system calls and text messages) (Bryce and Memon, 2007). Common examples of EDS include Graphical User Interfaces (GUIs), web applications, network protocols, embedded software, software components and device drivers, among which Graphical User Interface (GUI) plays a vital role in improving the Human-Computer Interaction (HCI) and plays a key role in the acceptability of the software. GUI consists of graphical controls that the user can select using mouse or keyboard and consists of components like menu bar, toolbar, windows and buttons. It has become the de facto standard for user interface in almost all of the modern technologies.

Research has shown that in general, 40-60% of the total software code has been used for implementing GUI (Myers, 1995). In spite of GUI providing easy way to use the software, they make the development process of the software complex and make up a large proportion of all software errors. All these make GUI testing a mandatory process where the goal is to ensure that the GUI meets its written specifications.

In spite of these studies showing the importance of testing in GUI, approaches that test the functional correction of these interfaces has been largely neglected and is only, in the past few years have got attention. A review of various techniques and metrics available for this purpose can be obtained from Vijayakumar and Punithavalli (2009).

General tests are not applied directly to GUIs because of the increased number of states generated because of huge number of permutations of input events. For adequate testing, an event may need to be tested in many of these states, requiring large number of test cases (each represented as event sequent) (Memon, 2007). This increases the need for reduction and prioritization of GUI test suites. In response to these requirements, this research presents a test case framework that focus on three important tasks, namely; GUI test case generation, reduction and prioritization.

Automation of these tasks are becoming complex as the GUI's are becoming more complex and the 21st century demands of end users require sophisticated user interfaces which further complicates the process of testing the correctness and underlying software. In event-driven architecture, user actions create huge number of events and the automatic test cases have to simulate these events. The high number of events makes the testing process a time consuming process. Large space of possibilities, as the user may click on any pixel on the screen and even the simplest components have a large number of attributes and methods.

GUI provides enormous amounts of possible interaction which again makes the process of testing an intricate job. To address these challenges, techniques that focus on optimizing the test case generation process along with reduction and prioritization techniques need to be designed. This study is an attempt made in this direction.

Corresponding Author: E. Vijayakumar, School of Computer Applications, Professional Institutions, Coimbatore, India

MATERIALS AND METHODS

The research proposes various techniques that are designed to find optimized solutions to the above questions and are grouped into 3 phases. The techniques proposed in each of these phases are described as:

Test case generation technique: Phase I of the study aims to automate the process of test case generation for GUI by incorporating additional characteristics that reduces the number of test cases generated. To achieve this aim, the Automatic Test Case Generation (ATCG) algorithm is enhanced to identify three types of user intereaction handlers, namely; shared event, context sensitive event and hidden widget handlers. The shared event handler represents common code fragments that are used by different user interactions while the context sensitive event handlers states that the control flow of a user interaction handling of a program fragment depends on the order of the preceding user interactions (Arlt *et al*., 2011). A hidden widget is a shortcut connected to a certain event handler in the same manner as the event handler of a button's click. Since, shortcuts that can be used by a user are not visible to them, mining the source code can reveal event handlers that are called if a particular key stroke occurs. Thus, the ATCG model generates test cases using the steps given as:

- Extract widgets and handlers
- Identify and eliminate shared events
- Identify context-sensitive events
- Identify hidden events
- Generate test cases

In this model, an application is defined by a GUI and a set of instructions (java code). The GUI consists of widgets (buttons, text boxes, radio buttons, etc.) which the users use for interaction. Each interaction generates an event (e) which consists of the widget used along with the type of interaction and each event is associated with an event handler (h). Event handlers are routines that are executed when an event (e) occurs. If E is the set of all events, H consists of all event handlers, the relation Ex: E×H can identify the set of instruction h = Ex (e) that handles an event e. The GUI test case is then a sequence of events t = {e1, ..., en} and an oracle descries if the output of a sequence meets the requirements. The GUI test model is defined as M = (s, δ) where S is a finite set of states and $\delta = S \times (E \cup \{\in\}) \times S$ is a set of transitions between two states labeled with an event $e \in E$.

For generating the model of the GUI, information about the possible user interactions from the source code is mined. The application is analyzed to collect all events (or user interactions) together with the source code fragment that handles this event. The mining procedure iterates over a JAVA program given as a set of classes. For each class, an iteration is performed over the attributes. For each attribute, a check is made to decide if the attribute is an instance of a widget type that appears in the GUI. If the attribute is not a widget type, it is skipped and the procedure continues with the next attribute. If the attribute is a widget type, the algorithm iterates over the event handlers associated with this widget. All events created for one widget are collected.

For each attribute in the source code that represents a widget, a pair of an event (e) that represents the interaction with this widget is recorded along with the program fragment h that handles this event. The fragment h is obtained by following the control-flow starting from the location that handles e. If the pair (e', h) is already recorded and if the events e and e' are handled by the same source code fragment h, dismiss the pair (e, h), as the event (e') is sufficient to test h. In the special case that h does not have control-flow (that is the event is not implemented), the pair (e, h) is dismissed again. In the resulting set, each event (e) is associated with a unique source code fragment h.

From the list of pairs of events and event handlers, the context of each event handler is next identified. Given a pair (e, h) of event and handler, the code (h) that handles an event (e) is a said to be a context sensitive event handler if the control-flow path taken when e is handled depends on variables that are modified by other event handlers. That is, if the control-flow path taken in h is influenced by certain preceding events. These events are identified by analyzing the control flow and for each conditional choice (i.e., loops, if-then-else, etc.), the algorithm checks if the condition evaluates attributes of objects that are derived from a JComponent. If such an object is detected, the events associated with this widget are added to the list of context events Ctx (h) of the handler (h). The hidden widgets and their corresponding events are identified by performed the search operations to include action listeners along with key and mouse listeners.

The automation model for test case generation is then built using the a list of pairs (e, h) of event (e) and event handler (h) along with list of events Ctx (h) that might affect the control-flow of h and hidden event handlers. The input of the test case generation algorithm includes the model, TC size that determines the number of events of each test case, timeout that determines for how long the test cases will be generated and the seed that determines the random seed used for test case generation.

The test case generation algorithm starts with an empty sequence of events tc. The testing procedure is repeated until a given timeout is reached. Starting from the initial state in model M, in each iteration of the loop, one outgoing edge is randomly picked and traverse to the target state. The event from the edge label is added to tc. This procedure is repeated until the initial state of M is reached again and the length of tc is larger than the threshold TC size. The resulting sequence of events tc is executed and a check is performed to find if the program terminates abnormally. If that be the case, this sequence is reported to test engineer and the sequence of events is reset and the process is started over until the time limit is reached.

Reduction of test cases: This phase aims to eliminate redundant events to reduce the number of test cases generated. As the number of test cases generated have a direct influence on the performance of the testing process, it is always desirable to reduce this number in a way it does not degrade the testing performance. For this purpose, this phase uses a 2-stage method to identify feasible and infeasible test cases and eliminates the infeasible test cases, thus reducing the number of test cases. Detailed description of the steps involved during test case generation while using the reduction technique is published by Vijayakumar and Punithavalli (2013).

Test case prioritization: Prioritizing and scheduling test cases are one of the most critical tasks during the software testing process. According to Gove and Faytong (2012), if there are approximately 20,000 lines of code, running the entire test cases requires around 7 weeks. In these type of situations, test engineers may want to prioritize and schedule those test cases in order that those test cases with higher priority are executed first. Test case prioritization techniques prioritize and schedule test cases in an order that attempts to maximize some objective function. Several researchers have focused on the problem of optimizing the operation of prioritization to identify the various advantages for improving testing process. However, the field still faces the following major issues:

- Existing test case prioritization methods ignore the practical weight factors in their ranking algorithm
- Existing techniques have an inefficient weight algorithm
- Pair-wise comparison has been successfully utilized in order to prioritize test cases by exploiting the rich, valuable and unique knowledge of the tester. However, the prohibitively large cost of the pairwise comparison method prevents it from being applied to large test suites

This phase of the study solves the earlier issues by using an Agglomerative Hierarchical Clustering Algorithm (Yoo *et al*., 2009) based on their runtime behaviour to reduce the required number of pair-wise comparisons significantly. The reduced number of required comparisons makes it feasible to apply expert-guided prioritization techniques to much larger data sets. The clustering process partitions objects into different subsets so that objects in each group share common properties. The clustering criterion determines which properties are used to measure the commonality. When considering test case prioritization, the ideal clustering criterion would be the similarity between the faults detected by each test case. However, this information is inherently unavailable before the testing task is finished. Therefore, it is necessary to find a surrogate for this, in the same way as existing coverage-based prioritization techniques turn to surrogates for fault-detection capabilities. In this phase, the dynamic execution traces of each test case are used as a surrogate for the similarity between features tested. Execution of each test case is represented by a binary string. Each bit corresponds to a statement in the source code. If the statement has been executed by the test case, the digit is 1; otherwise it is 0. The similarity between two test cases is measured by the distance between 2 binary strings using Hamming distance.

Prioritization of a clustered test suite is a different problem from the traditional test case prioritization problem. The two separate layers of prioritization are required in order to prioritize a clustered test suite (Roongruangsuwan and Daengdej, 2010). Intra-cluster prioritization is prioritization of test cases that belong to the same cluster whereas inter-cluster prioritization is prioritization of clusters. The study combines both intra and inter-cluster prioritization and the method is termed as Combined Intra and Inter Clustering Prioritization (C2ICP) Algorithm. In C2ICP, intra-cluster prioritization is performed first. Based on the results of intra-cluster prioritization, each cluster is assigned a test case that represents the cluster. Using these representatives, C2ICP performs inter-cluster prioritization.

Apart from the earlier cluster-based prioritization algorithm, phase III method also considers four types of techniques, namely; customer requirement based, coverage based, cost effective based and chronographic history based technique. These are included to solve the problems of same weight being assigned to >1 test case and multiple test suites. During prioritization of each cluster, first the defect factor is used to weight the test cases. In case of test cases having same weights, the time factors are considered. If further analysis, still produce same weights then cost factors are considered to calculate the weight. If the test cases still have same weights then finally, the weights are assigned base on text execution history. Finally, the intra and inter clusters are arranged in descending order of test case weights and the test case with highest priority is executed first. This technique presents the advantage of improving the capability of ranking or scoring test cases when there are multiple test cases with the same priority. This procedure increases the efficiency of ranking test cases and improves the GUI testing process.

Further, the proposed method is further enhanced to handle prioritization when there are >1 test suites, each having a set of test cases. This enhancement operation first prioritizes the test cases inside each suite separately and then prioritizes the test suits according to the summation of the cumulative product of weights and value of each priority technique.

Proposed framework: The proposed enhanced GUI test case generation algorithm, thus performs the techniques proposed in phase I-III of the study in a sequential manner to perform GUI testing. Experimental results showed that the 2-stage classifier that uses wavelet neural network for stage 1 and support vector machine for stage 2 produced better results and therefore are used in the framework.

RESULTS AND DISCUSSION

Several experiments were conducted to analyze the performance of the proposed algorithms. The experiments were conducted in four groups with each group of experiments focusing on each of the study. The algorithms were tested using TerpOffice applications, TerpWord, TerpPaint TerpPresent and TerpSpreadsheet (TerpOffice, 2009). The enhanced automatic generation of test cases algorithm was analyzed using the number of test cases generated. Table 1 shows the effect of test case generation algorithm on number of test cases generated. From Table 1, it is evident that the number of test cases generated while using the shared and context sensitive event handler is small.

The 2nd phase algorithms were analyzed using three parameters, namely; fault detection rate and error rate of identifying the errors. The 9 combinations of classifiers were developed by varying the three classifiers (BPNN, SVM and WNN) used by stage 1 and 2 of the algorithm, nine combinations of classification models were developed to reduce number of test cases generated. They are BPNN + BPNN (BB), SVM + SVM (SS), WNN + WNN (WW), BPNN + SVM (BS), SVM + BPNN (SB), BPNN + WNN (BW), WNN + BPNN (WB), SVM + WNN (SW) and WNN + SVM (WS). Table 2 and 3 present the detection and error rates of the classifiers. The percentage of training data used for training stage 1 classifier is 80%. The results obtained are compared with conventional classifiers SVM, WNN and BPNN.

From phase II results, it can be seen that all the 9 proposed models have improved the conventional classification method and the combination that used WNN for stage 1 and SVM for stage 2 is the winner among the 9 proposed classification model.

The 3rd phase algorithms were compared using high priority reserve effectiveness and size of acceptable test cases. The results obtained are presented in Fig. 1 and 2. From the results, it is evident that the proposed algorithm has improved the process of test case prioritization and thus, the overall testing with all the selected applications.

Table 1: Number of test cases generated

Applications	No. of test cases with shared vent handler	No. of test cases with context-sensitive event handler	No. of test cases with shared and context-sensitive event handler
TerpWord	48	29	15
TerpPaint	98	56	41
TerpPresent	67	47	22
TerpSpreadsheet	83	39	25

Table 2: Detection rate (%)

Terp application	BB	SS	WW	BS	SB	BW	WB	SW	WS	SVM	WNN	BPNN
Word	91.82	98.44	98.69	91.99	97.99	92.96	98.15	98.71	98.77	90.83	91.36	89.15
Paint	88.11	94.32	95.08	88.24	94.02	88.59	94.31	95.14	95.50	86.04	87.03	84.72
Present	90.06	96.75	97.18	90.51	95.90	91.01	95.99	97.34	97.82	87.91	89.06	86.57
Spreadsheet	88.96	95.65	96.43	89.44	94.78	89.94	94.96	96.50	96.71	86.75	87.47	85.14

Table 3: Error rate (%)

Terp application	BB	SS	WW	BS	SB	BW	WB	SW	WS	SVM	WNN	BPNN
Word	2.11	0.12	0.11	1.35	0.29	1.26	0.24	0.11	0.11	7.08	5.77	8.46
Paint	4.05	1.05	0.80	3.11	1.75	2.58	1.50	0.50	0.39	9.96	8.05	10.91
Present	5.62	1.88	1.44	5.14	2.28	4.64	2.22	0.95	0.92	11.04	9.89	12.00
Spreadsheet	6.87	2.63	2.07	5.73	3.81	5.49	3.45	1.67	1.42	13.16	11.49	13.20

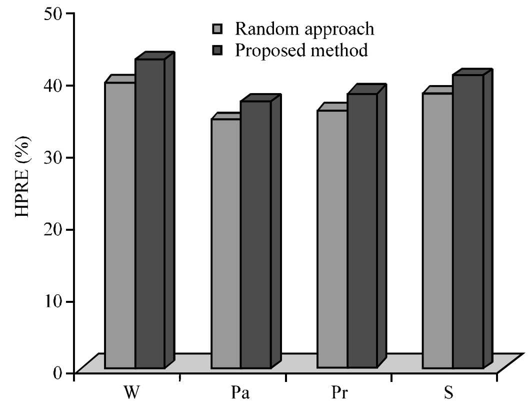

Fig. 1: High priority reserve effectiveness (%)

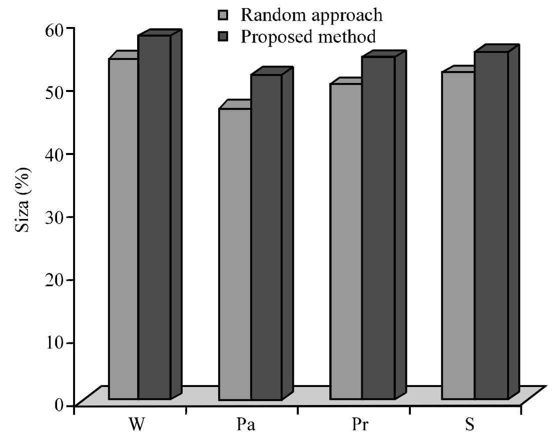

Fig. 2: Size of acceptable test cases (%)

Table 4: Fault detection rate (%)

Framework	Word	Paint	Present	Spreadsheet
Test Case Generation (TCG)	89.26	90.27	92.08	93.84
TCG with reduction	98.77	95.50	97.82	96.71
TCG with reduction and prioritization	99.25	96.40	98.58	97.90

Table 5: Error rate in Terp applications (%)

Framework	Word	Paint	Present	Spreadsheet
Test Case Generation (TCG)	0.97	1.15	1.54	2.94
TCG with reduction	0.11	0.39	0.92	1.42
TCG with reduction and prioritization	0.08	0.22	0.80	1.26

The last stage of experimentation analyzes the test case framework that is build using the winning algorithms of each phase. The performance metrics used are analyzed using detection rate and error rate and the results obtained are tabulated in Table 4 and 5. Experimental results prove that the techniques proposed in each phase enhances the performance of its respective process and the framework that combines these algorithms improves the fault detection of the GUI applications.

CONCLUSION

This research is focused on improving the GUI testing process and performs it using three main steps, namely; test case generation, selection and prioritization. The methodology framed proposes techniques for each of these techniques which are then combined to form a automated GUI testing framework. The experimental results proved that the performance of the proposed models at each step improved the performance of the existing algorithm. Further, the results also proved that the proposed fault detection frame is successful in identifying errors in GUI and can be used by software industries to effectively identify errors and to improve overall software quality.

REFERENCES

Arlt, S., C. Bertolini and M. Schaf, 2011. Behind the scenes: An approach to incorporate context in GUI test case generation. Proceedings of the IEEE 4th International Conference on Software Testing, Verification and Validation Workshops, March 21-25, 2011, Berlin, Germany, pp: 222-231.

Bryce, R.C. and A.M. Memon, 2007. Test suite prioritization by interaction coverage. Proceedings of the Workshop on Domain Specific Approaches to Software Test Automation: In Conjunction with the 6th ESEC/FSE Joint Meeting, September, 4, 2007, Dubrovnik, Croatia, pp: 1-7.

Gove, R. and J. Faytong, 2012. Identifying infeasible GUI test cases using support vector machines and induced grammars. Proceedings of the IEEE 4th International Conference on Software Testing, Verification and Validation Workshops, March 21-25, 2011, Berlin, Germany, pp: 202-211.

Memon, A.M., 2007. An event-flow model of GUI-based applications for testing. J. Software Test. Verification Reliab., 17: 137-157.

Myers, B.A., 1995. User interface software tools. ACM Trans. Comput. Hum. Interact, 2: 64-103.

Roongruangsuwan, S. and J. Daengdej, 2010. Test case prioritization techniques. J. Theor. Applied Inform. Technol., 18: 45-60.

Vijayakumar, E. and M. Punithavalli, 2009. A survey on user interface defect detection in object oriented design. Global J. Comput. Sci. Technol., 9: 176-182.

Vijayakumar, E. and M. Punithavalli, 2013. Enhanced GUI test case generation method using two-stage classification method. Int. J. Comput. Appl., 63: 29-33.

Yoo, S., M. Harman, P. Tonella and A. Susi, 2009. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. Proceedings of the 18th International Symposium on Software Testing and Analysis, July 19-23, 2009, Chicago, IL, USA., pp: 201-212.