Enhanced GUI Test Case Generation Method using Two-stage Classification Method

E.Vijayakumar,
Asst.Professor/MCA,
Professional Institutions,
Coimbatore,Tamilnadu,India

M.Punithavalli,
Director/MCA,
Ramakrishna Engg.College,
Coimbatore, Tamilnadu,India

ABSTRACT

Software testing is a task of quality assurance where the main aim is to identify errors. Graphical User Interfaces (GUIs), a class of Event-Driven Software (EDS), is increasingly used to increase the human-to-computer interaction. 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. This paper proposes techniques that use a reduction-based test case generation model that is enhanced by identifying feasible and infeasible test sequences. The proposed method uses a two-stage classification process, where two classifiers, BPNN (Back Propagation Neural Network) and Support Vector Machine (SVM), are used. The main goal here is to improve the performance of the second classifier, SVM, by using the results of the first classifier, BPNN. Experimental results show that the proposed method has increased the accuracy of classification.

Keywords

Graphical User Interface, Test Case Generation, Infeasible Test Sequences, Support Vector Machine, Back Propagation Neural Network.

1. INTRODUCTION

Advancements in information technology, computers and intelligent devices have increased the use of software systems in all aspects of modern society. This increased use of software in daily life demands them to function without errors. Quality assurance is the planned and systematic way of monitoring the methods and processes of a software to ensure quality. One popular quality assurance technique is Software Testing which is the planned process that is used to identify the correctness and completeness of a software system and is employed during the development, implementation and maintenance phases of a software lifecycle.

Event-Driven Software (EDS) have rapidly become a critical part of business for 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 Menon, 2007) Common examples of EDS include graphical user interfaces (GUIs) (Abdul et al., 2010), web applications (Kumar and Goel, 2012), network protocols (Gong *et al*., 2009), embedded software (Gu and Shin, 2005), software components (Adams and Grib, 1999) and device drivers (Tchamgoue *et al*., 2012). The term Events can be user actions such as clicking a mouse button or pressing a key or System occurrences. Most Modern EDS applications, particularly those that run in Macintosh and Windows environments, are said to be Event-Driven because they are designed to respond to events. Quality assurance tasks (testing) have become important for EDS as they are increasingly being used in many important applications.

One important class of EDS is Graphical User Interface (GUI) which is used to improve the Human-Computer Interaction (HCI). Graphical User Interface (GUI), consists of graphical controls that the user can select using mouse or keyboard and typically, consists of components like menu bar, toolbar, windows and buttons and has become the de factor standard for user interface in almost all of the modern technologies. Research has shown that, in general, 40% to 60% of the total software code has been used for implementing GUI (Memon, 2007; Myers, 1995). In spite of GUI providing easy way to use the software, they make the development process of the software complex (Isabella and Retna, 2012) and make up a large proportion of all software errors. A case study conducted by Mohapartra (2001) that investigated the sources of errors using a live project in INFOSYS Technologies Limited, India, revealed that more than 50% of errors were contributed by GUI alone. 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.

GUI testing consists of methods for validating GUI objects, checking functional flows by operating GUI objects and verifying output data which are generated in backend and then displayed in front pages (Xiaochun *et al.*, 2008). GUI testing can be performed manually or in a semi-automatic or fully automatic fashion. However, the tendency is to automate as much as possible so as to make it very fast and have a huge coverage which would otherwise take a tremendous time for a human. Several researchers have proposed strategies for automating the GUI testing (Zhao, 2006; Hendrick *et al.*, 2005) as they have the potential to reduce testing cost and improve software quality.

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; Khum *et al.*, 2004). This increases the need for reduction and prioritization of GUI test suites. In response to these requirements, this paper presents a test case framework that focus on three important tasks, namely, GUI test case generation, test case prioritization and test case reduction.

Arlt *et al.* (2011) proposed a GUI test case generation method that reduced the number of test cases generated by identifying two user interactions, namely, shared event handlers and context-sensitive event handlers. The shared event handler represents common code fragments that are used by different user interactions. 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. This framework eliminates redundant events and thus reduces the number of test cases generated. As the number of test cases generated has a direct influence on the performance of the testing process, it is always desirable to reduce this number in a way it does degrade the testing performance. For this purpose, this paper uses a method to identify feasible and infeasible test cases and avoids infeasible test cases. An event sequence in a test case is infeasible when atleast one event that is expected to be available at the point during execution is not available by the GUI state. This situation may arise due to a bug in GUI or a constraint between events in GUI specification.

The identification of feasible and non-feasible test cases is done using a novel two-stage machine learning classification algorithm that uses two different classifiers during discovery of infeasible events. The aim of using two classifiers is: given an test dataset, T, consisting of test sequences $\{t_1, t_2, \ldots\}$, the aim of the first classifier is to preprocess T for data reduction, that is, the first classifier identifies all correctly classified data to obtain a refined dataset T' of T. T' is then used to train the second classifier which identifies feasible test cases. The usage of the refined dataset could improve the classification performance of the second classifier. Usage of machine learning algorithm in optimizing test case generation process is sparse (Gove and Faytong, 2012) eventhough its use in various other segments of software engineering is vast. The use of two classifiers, to the best of author's knowledge, is a new concept that is new in software testing. The rest of the paper is organized as follows. Section 2 describes the framework proposed by Arlt *et al.* (2011). This model is referred to as ATCG (Automatic Test Case Generation) model in this paper. Section 3 presents the methodology used to enhance ATCG, which combines the identification of infeasible test cases using two-stage classifiers. Section 4 presents the experimentation results while Section 5 concludes the research work with future research directions.

ATCG MODEL

The ATCG model generates test cases in three steps (Figure 1). They are, extracting widgets and handlers, generating the test model and generate and execute 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 widge 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 x 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 $t = \{e_1, \ldots, e_n\}$ and an oracle descries if the output of a sequence meets the requirements. The GUI test model is defined as $M = (s,\delta)$, 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 $e \in E$.

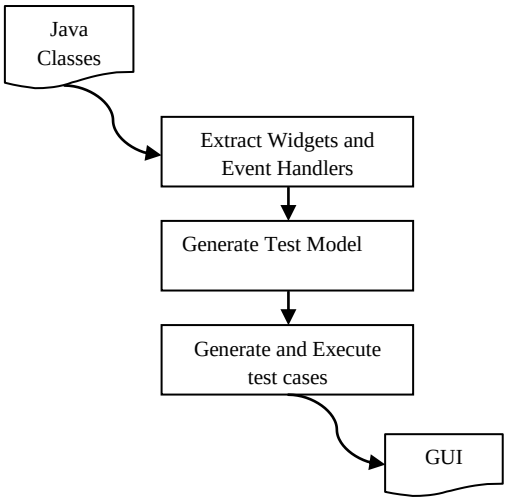


Figure 1 : Test Case Generation Model

The pseudo codes for extracting widgets and handlers, collecting context-sensitive event handlers and test case generation are given in Figures 2 to 4 respectively. The first step produces a list of (e, h) of event e and event handler h, as well as a list of events Ctx(h) that might affect the control-flow of h. The model $M = (S, \delta)$ is a finite automaton with transitions labeled with events e or ϵ. First, the model contains an initial state $s_0 \in S$ and no transitions. For each pair (e, h), a state s and one transition (s_0, e, s) and another transition (s, ϵ, s_0) that loops back to the initial state is created. The process iterates over the set of context events Ctx(h) and finally, a new state s' and an edge (s_0, ec, s') is created for each $ec \in$ Ctx(h) and one edge (s', e, s).

```
for each (Java class in set) do
   for each (attribute in Java Class) do
      if (attribute is a widget) then
         for each (Event e of widget) do
            if (e is in Ex ) then
               Skip (redundant) widget
            else if (event handler h is empty) then
               Skip (dead) widget
            else
               Store Ex (e) = h
            end if
         end for
      else
         Skip attribute
      end if
   end for
end for
```

Figure 2 : Extract Widgets and Handlers

```
State s = initial state of M; Test Case tc := {}
while (timeout is not reach) do
   Pick randomly (s, e, s') ∈ δ; s := s′; tc := tc + e
   if (size(tc)≥TCsize) and (s is initial state) then
      try {Execute the test case tc}
         catch {Report tc }
      end try
      Test Case tc := {}
   end if
end while
```

Figure 3 : Context-Sensitive Event Handler

```
Begin
   Ctx (h) = Φ;
   for each (conditional choice c in h) do
      if (c reads field of widget w) then
         Add events of w to Ctx (h);
      end if
   end for
end
```

Figure 4 : Test Case Generation

3. PROPOSED METHOD

The proposed model consists of four steps while creating the GUI test case model. The first two steps correspond to the ATCG model. The third step is modified to include a reduction step that identifies infeasible event sequences. For this purpose, the model uses Back Propagation Neural Network (BPNN) and Support Vector Machine (SVM). The classifiers considered are binary classifiers as the test sequences have to be grouped as either feasible or infeasible. The steps involved in step 3 are summarized in Figure 5.

Step 1 : Generate test sequences using procedure in Figures 2, 3, 4.

Step 2 : Conversion of test sequences – Classifiers require data to be numeric vectors. But test cases are sequences of ID. Hence, a conversion process is needed. Let the converted test sequence be termed as dataset, T.

Step 3 : Partition T into training (Tr) and testing (Te) sequences.

Step 4 : Use Tr to train BPNN and test the trained network using Te. Collect only those sequences of T that were correctly classified and **treat this new testing set (Te').**

Step 5 : Use Te' to train the second classifier SVM.

Figure 5 : Test Case Generation and Reduction Algorithm

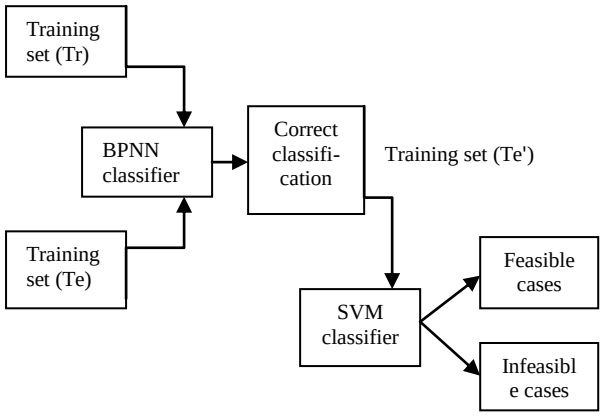
The conversion process mentioned in step 2 consists of four stages. The first stage handles the value assignment of the first N vector attributes, where N is the number of different IDs in the sequence. Index 'i' in the resulting vector corresponds to the number of times that event 'i' appears in the test case. This step simply counts the number of appearance of each ID. The second stage determines all the possible pairwise combinations. For this purpose, a vector P is created which lists all the possible combinations of the available IDs by iterating from the lowest ID to the highest. Combinations of the same ID are excluded (example : 22). Each event ID 'i' will have N − 1 ordered pairs that start with ID 'i', where N is the number of different IDs in the GUI. The third stage then iterates through the input test case counting all the appearances of each combination found in vector P (e.g., counting the number of times that 0 occurs immediately before 1, the number of times 0 occurs immediately before 2, etc.). Each count is appended to the Basic vector. Finally, the fourth stage iterates through the test case counting all the appearances of each generated combination as well, with one difference to the third stage: this run counts all the times the second pair ID appears after the first pair ID in the given test case. Each of these results is appended to the Basic vector.

For example, let a GUI have event IDs {0, 1, 2} and let the original test case be <0 1 2 1 2>. Then the converted feature vector is after stage 1 is <1 2 2>. This stage simply counts the number of appearance of each ID. Stage 2 produces < 01 02 10 12 20 21 > and Stage 3 results with <1 2 2 1 0 0 2 0 1> where <1 2 2> is the original basic vector from step 1 and <1 0 0 2 0 1> are the counts of each pair in P that occurs in the test case. The final converted feature vector is produced by stage 4 is <1 2 2 2 2 0 2 0 1>.

In the next step, the converted feature vector is partitioned into training and testing sets. In order for the classification algorithms to learn to classify feasible and infeasible test cases, the training data must include both types of test cases. Let Tr and Te denote the training and testing partitions. A constant x is included to indicate the number of data used for training. The paper uses five values for x, {10, 20, 30, 40, 50).

The partitioning process begins by constructing two sets I and F, each consisting of feasible and infeasible test cases respectively. Add x% of I and F to Tr and the remaining to Te.

The final step uses the two-stage classifiers to classify the test cases as feasible and infeasible. The process is shown in Figure 6.



4. EXPERIMENTAL RESULTS

The procedure for creating the test case is detailed as follows. As mentioned previously, the data set is a set of test suites, with each test suite consisting of a set of test cases. Each test case is of length n and is composed of a string containing n tokes denoting the GUI events. The tokens can be number (test case), feasibility (F) or a value 0 indicating a failure status. For example, consider <1 0 3 2 4 F 3>. This is a test case sequence of length 5 and is considered infeasible and failed on event at index 3, corresponding to event 2. The test case length considered as 5, 10, 15 and 20. To evaluate the performance of the proposed method, four Terp applications (TerpOffice, 2009) namely, TerpWord, TerpPaint TerpPresent and TerpSpreadsheet was used. The attributes of the selected applications are given in Table 1.

Attribute	Terp Word	Terp Paint	Terp Present	Terp SpreadSheet
Classes	9	42	4	25
Event Handlers	39	95	91	37
Shared Event Handler	11	1	24	10
Context Sensitive Event Handler	24	69	74	35
Windows	8	8	5	6
Widgets	50	92	115	51
LOC	6842	17730	25072	126909

The main aim of the experiments was to analyze the effect of training data set size on classification errors. For this purpose, the training set size was creating using 10%, 20%, 30%, 40% and 50% of the GUI test cases for each GUI application. The correctly classified percentage is used as the performance metric to analyze the performance of the proposed method. The results are compared with single classifier (SVM). Figures 7 to 10 shows the results obtained for the four applications respectively.

From the figures, it could be seen that with all the four applications, the proposed two-stage classifier that uses two classifiers, BPNN and SVM, shows improved performance. On average, the proposed method showed 84.65%, 85.52%, 87.52% and 82.47% classification accuracy. When compared with the average result of the existing (SVM) classifier, the proposed performance showed an average gain of 4.29%, 3.32%, 3.47% and 4.04% respectively on TerpWord, TerpPaint, TerpPresent and TerpSpreadsheet. While considering the varying training size, it can be seen that the classification performance has a direct impact on training size.

5. CONCLUSION

This paper presented an enhanced test case model that first created test cases by reducing the number of redundant test cases. Further the test cases generated were further refined by identifying infeasible test sequences. For this purpose, a two-stage classifier was used. The two stage classifier uses two classifiers, namely, BPNN and SVM. The BPNN classifier was first used to classify the test case dataset and only the corrected classified data was used to train the second classifier, SVM. This two-step process, improved the classification efficiency in terms of correctly classified data. On average, the proposed method was able to achieve an accuracy of 84.21%, which is a positive improvement when compared with the average performance of 82.73% when classified with SVM classifier. In future, plans to enhance this model with prioritization techniques are made.

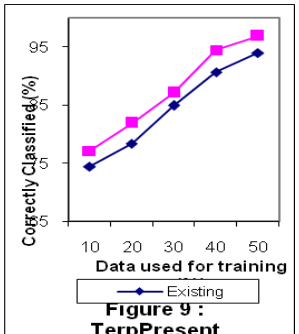


Figure 9 : TerpPresent

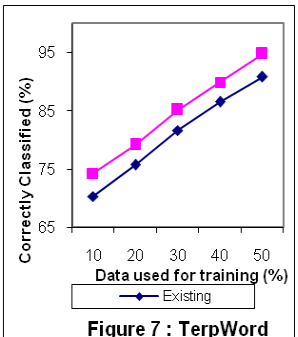


Figure 7 : TerpWord

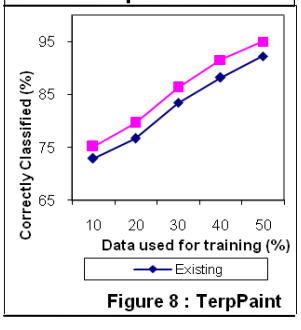


Figure 8 : TerpPaint

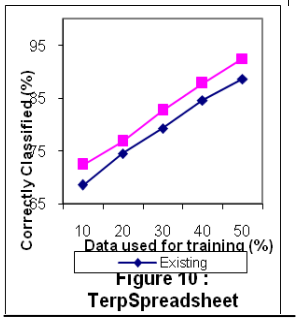


Figure 10 : TerpSpreadsheet

6. REFERENCES

[1]. Abdul, R., Naveed, E., Qamar, A., Shafiq, R. and Ali, S.A. (2010) PSO based test coverage analysis for event driven software, 2nd International Conference on Software Engineering and Data Mining (SEDM), Pp. 219-224.

[2]. Adams, M.M. and Grib, T.E. (1999) A component based, event driven framework for rapid prototyping real-time avionics systems, Proceedings. 18th Digital Avionics Systems Conference, Vol. 2, Pp. 9.C.5-1 - 9.C.5-8.

[3]. Arlt, S., Bertolini, C. and Schaf, M. (2011) Behind the Scenes: An Approach to Incorporate Context in GUI Test Case Generation, Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW '11). IEEE Computer Society, Washington, DC, USA, Pp. 222-231.

[4]. Bryce, R.C. and Menon, A.F. (2007) Test Suite Prioritization by Interaction Coverage, Proceeding of Workshop on Domain specific approaches to software test automation in conjunction with the 6th ESEC/FSE joint meeting, ACM, New York, Pp. 1-7.

[5]. Gong, H., Liu, M., Yu, L. and Wang, X. (2009) An Event Driven TDMA Protocol for Wireless Sensor Networks, WRI International Conference on Communications and Mobile Computing, Vol. 2, Pp. 132-136.

[6]. Gove, R. and Faytong, J. (2012) Identifying Infeasible GUI Test Cases Using Support Vector Machines and Induced Grammars, IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Pp. 202 - 211

[7]. Gu, Z. and Shin, K.G. (2005) Model-checking of component-based event-driven real-time embedded software, Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, Pp. 410-417

[8]. Hendrick, S.D., Hendrick, K.E. and Webster, M. (2005) Worldwide Distributed Automated Software Quality Tools 2005–2009 Forecast and 2004 Vendor Shares, IDS Software Research Group.

[9]. Isabella, A. and Retna, E. (2012) Study paper on test case generation for gui based testing, International Journal of Software Engineering and Applications (IJSEA), Vol.3, No.1, Pp. 139-147.

[10]. Kuhn, D.R., Wallace, D.R. and Gallo, A.M. (2004) Software fault interactions and implications for software testing, IEEE Transactions on Software Engineering, Vol.30, No.6, Pp.418–421.

[11]. Kumar, A. and Goel, R. (2012) Event driven test case selection for regression testing web applications, International Conference on Advances in Engineering, Science and Management (ICAESM), Pp. 121-127.

[12]. Memon, A.M. (2007) An event-flow model of GUI-based applications for testing, Journal of Software Testing, Verification and Reliability, Vol. 17, Issue 3, Pp. 137-157.

[13]. Myers, B.A. (1995) User Interface Software Tools, ACM Transaction Computer Human Interact, Volume 2, Issue 1, Pp. 64–103.

[14]. Tchamgoue, G.M., Kim, K.H. and Jun, Y.K. (2012) Testing and Debugging Concurrency Bugs in Event-Driven Programs, International Journal of Advanced Science and Technology, Vol. 40, Pp. 55-68.

[15]. TerpOffice(2009),http://www.cs.umd.edu/~atif/TerpOfficeWeb, Last Access Date : 01-01-2013.

[16]. Xiaochun, Z., Bo, Z., Juefeng, L. and Qiu, G. (2008) A test automation solution on GUI functional test, Proceedings of the 6th IEEE International Conference on Industrial Informatics, INDIN, Pp. 1413–1418.

[17]. Zhao, N.Y. and Shum, M.W. (2006) Technical Solution to Automate Smoke Test Using Rational Functional Tester and Virtualization Technology, Computer Software and Applications Conference, COMPSAC '06, Vol. 2, Pp. 367–367.