Selecting Test Cases by Cluster Analysis of GUI States

Yang Gao, Cheng-Gang Bai

***Abstract*—Nowadays graphical user interface (GUI) has been widely used in software systems, while there is no efficient testing techniques for the rapidly evolving GUI applications. For the GUI applications are modified rapidly and the test suites trend to be huge in size, it is often desirable to select a subset of test cases to fulfill the regression testing. In this paper, a novel GUI state model is presented to address the execution of test case, and then a state-coverage method based on cluster analysis of the GUI states is proposed to select a reliable subset of test cases for GUI regression testing. An empirical study illustrates that the state-coverage method is effective for GUI test case selection.**

I. Introduction

It is often desirable to select a subset of existing test suite to test or evaluate software's conformance to its requirement. In particular, Graphical User Interface (GUI) applications are created using rapid prototyping and modified frequently [1], thus continual regression testing is performed to instill confidence of the changes during the software development and maintenance. GUI is typical event-driven software (EDS), and takes event sequences as input, which is quite different from traditional software. GUI can bring immense flexibility to users, but the large number of possible event sequences severely hinder the adequate testing of GUI software. Even after finding a potential subsection of event sequences related the modifications [2], the number of the test cases may be still too large to employ the retest-all approach.

Generally, the nature and location of faults are not known in advance, so some rules, like coverage criteria of the test suites, are utilized in order to estimate whether a program has been adequately tested and to guide the testing process. However, it has been proposed that conventional code-based coverage criteria, including statement coverage, branch coverage, and path coverage, are not suitable for GUI testing [3]. On one hand, code of GUI needs to be tested in different contexts. Taking common "save" command as example, two similar test cases <New a File; Input Something; Save> and <Open an Existing File; Input Something; Save> would execute quite different code. Hence two test cases both need to be tested. On the other hand, the excessive testing may consume extra time and resources, which is not expected in the quick regression testing procedure. Meanwhile, the existing test cases generation techniques prevailing based on Event-flow Graph (EFG), both manual and automated, in some sense, are of blind. Traversing the EFG or EFG+ [4] to obtain all the legal event sequences inevitably generate many repetitive or similar test cases. Consequently, a respective bug is repetitively detected by dozens possibly hundreds of test cases in regression testing [5] which is not expected for testers.

GUI state, modeled as sets of widgets, is used as test oracle in GUI testing [6]. Note that manually auditing is necessary after mismatches were found based on comparison test oracles, there is a severe need to conduct a further selection to fulfill the continual regression testing. To be worthwhile, the cost of selection process must be inexpensive. In fact, a significant feature of regression testing is that a test suite and profiles of execution for previous version of the software are available. Moreover, some studies [7][8] also indicated that distribution-based techniques can be as efficient or more efficient for revealing defects than coverage-based techniques. So it is reasonable to conduct a further selection with extensive coverage of execution state space that may be relatively reliable and efficient for a quick regression testing.

In this paper, we firstly give the definition of our particular GUI state to characterize the execution of test case, and then focus on the distribution of GUI states to select a representative subset of test cases. The selecting procedure is based on automatic cluster analysis of GUI states, and two different clustering methods are used and compared. To better address the relationship of GUI states, similarity of GUI states considering asymmetry of states is refined according to our previous work [9]. Empirical study on two open source software programs are conducted to illustrate that the test cases selected based on cluster analysis of GUI states are representative and efficient for revealing defects.

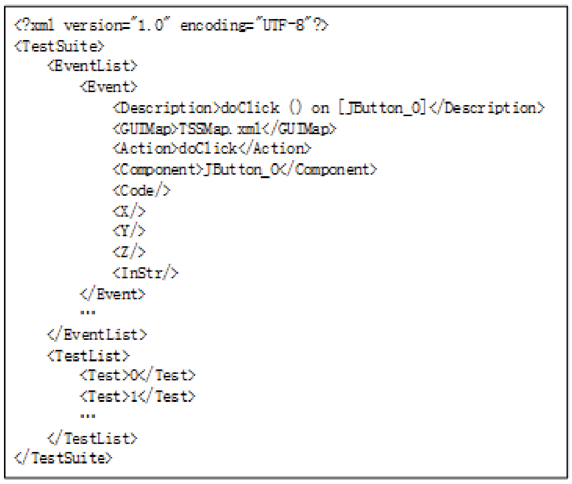
II. GUI State Model

A GUI is composed of windows and widgets which are visible for users to interact with. Due to the nature of EDS, GUI takes event as input, and changes its state. A GUI test case is often modeled as an initial state followed by a legal event sequence [10], such as $\langle S_0, e_1; e_2; \cdots e_n \rangle$, where S_0 is the initial state and event e_{i+1} can be performed immediately after e_i. In order to reduce the cost of execute all the test case, we assign the initial state of GUI application to S_0 and denote a test case as a single event sequence in this paper. We record our test suites in XML files which contain an event list and a test case list. Fig. 1 partially shows an example of test suite XML file. The event list contains several the events the GUI application can accept, and the test case list shows the event sequences of the test suite. The number consisting in the test case list presents an event according to the event list.

Yang Gao is with the Department of Automatic Control, Beihang University (Beijing University of Aeronautics and Astronautics), Beijing 100191, China (e-mail: gxjh_a@163.com)

C.-G. Bai is with the Department of Automatic Control, Beihang University (Beijing University of Aeronautics and Astronautics), Beijing 100191, China (e-mail: bcg@buaa.edu.cn)

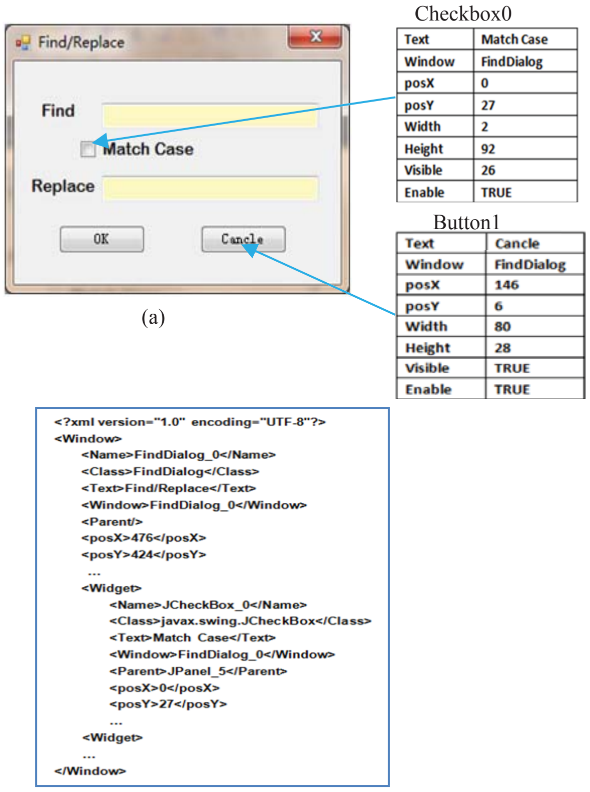
978-1-4673-8318-9/16/$31.00©2016 IEEE



```
<?xml version="1.0" encoding="UTF-8"?>
<TestSuite>
    <EventList>
        <Event>
            <Description>doClick () on [JButton_0]</Description>
            <GUIMap>TSSMap.xml</GUIMap>
            <Action>doClick</Action>
            <Component>JButton_0</Component>
            <Code/>
            <X/>
            <Y/>
            <Z/>
            <InStr/>
        </Event>
        ...
    </EventList>
    <TestList>
        <Test>0</Test>
        <Test>1</Test>
        ...
    </TestList>
</TestSuite>
```

Figure 1. Part of test case suite XML file

After event e_i was accepted, GUI application transform state S_i into S_{i+1}. The state sequence $\langle S_1, S_2 \cdots S_n \rangle$ contains all the run-time states which can directly and adequately reflect the execution. However the exorbitant cost to audit the run-time states prevents it from being widely applied in the real world. To efficiently characterize the execution, GUI state, for a test case in this paper, is refined as a synthesis of all the final states of windows ever invoked during the test execution.



Text	Match Case
Window	FindDialog
posX	0
posY	27
Width	2
Height	92
Visible	26
Enable	TRUE

Text	Cancle
Window	FindDialog
posX	146
posY	6
Width	80
Height	28
Visible	TRUE
Enable	TRUE

```
<?xml version="1.0" encoding="UTF-8"?>
<Window>
    <Name>FindDialog_0</Name>
    <Class>FindDialog</Class>
    <Text>Find/Replace</Text>
    <Window>FindDialog_0</Window>
    <Parent/>
    <posX>476</posX>
    <posY>424</posY>
     ...
    <Widget>
        <Name>JCheckBox_0</Name>
        <Class>javax.swing.JCheckBox</Class>
        <Text>Match Case</Text>
        <Window>FindDialog_0</Window>
        <Parent>JPanel_5</Parent>
        <posX>0</posX>
        <posY>27</posY>
        ...
    <Widget>
    ...
</Window>
```

Figure 2. (a) Find/Replace GUI (b) the XML file

GUI applications change their states by invoking different windows and setting widgets different properties. According to our previous work [9], GUI window can be modeled as a set of triples $S_{win} = \{(w, p, v) \mid w \in W_{win}, p \in P_w, v \in V_p\}$, where W_{win} is the set of widgets in window win, P_w is set of selected properties to address widget w, V_p is the set of possible values of property p. Fig. 2 (a) give an example of the $Find / Replace$ window which contains many widgets, such as: $Text\ Field$, $Check\ Box$, $Button$ etc. So GUI state is described as $S(tc) = \{S_{win_i} \mid win_i \in win(tc)\}$, where is the set of windows invoked during the execution of test case tc. Each window state are recorded in an XML file before the window is destroyed. Fig. 2 (b) partially shows the XML file of $Find / Replace$.

The similarity of states is calculated based on pair-wise comparison. For the convenience of comparison, we select 13 items to address all the widget, including "Name", "Parent", "Window", "Class", "Text", "PosX", "PosY", "PosOrder", "Width", "Height", "Visible", "Enable" and "Value". The first four property are used to identify whether the two widgets are the same widget under different states, the others are used to calculate the similarity of the states. The metric of calculation of similarity is defined like that:

If the property p of widget w has the same value at different states, the similarity of the p is 1, otherwise is 0, then sim_p (similarity of p) is represented as follows:

$$sim_p = \begin{cases} 0, v \neq v' \\ 1, v = v' \end{cases}.$$

The similarity of widget w (sim_w) is defined as the average of its sim_p s, then:

$$sim_w = \frac{1}{9} \sum sim_p .$$

The similarity of window win (sim_{win}) is computed as:

$$sim_{win} = \frac{1}{m} \sum sim_w .$$

Where the number $m = |W_{win} \cup W'_{win}|$, and the widget $w \in (W_{win} \cap W'_{win})$.

The similarity of final GUI state (sim_{state}) is described as:

$$sim_{state} = \frac{1}{t} \sum sim_{win} .$$

Where the number $t = |win(tc) \cup win(tc')|$, and the window $win \in (win(tc) \cap win(tc'))$.

III. Cluster Analysis of GUI State

Cluster analysis is a frequently used method for multivariate analysis in many different fields such as machine learning, data mining, image processing, and bioinformatics. In this work, GUI states is used to characterize test cases, and clustering process is adopted to divide the population of GUI states into different subsets so that test cases in the same set potentially go through similar run-time states and reveal similar sets of faults. Therefore, it is reasonable to take state-coverage as a surrogate for fault-detection capability. Here we define the distance between GUI states as: $Dis = 1 - sim_{state}$. Note that the similarity relationship may be sketchy and not always transitive, so the two main approaches, both partitioning and hierarchical clustering, are considered and compared.

The partitioning clustering approach initially selects k objects as initial clusters, and then iteratively assigns the rest objects into the k clusters. To improve the cluster quality, the k initial objects are selected as follows: firstly select state $s_1, s_2 \in S$ to make the maximum of Dis, where S is the GUI state space of the test suite; then iteratively select state $s_i \in S / \{s_1, s_2 \cdots s_{i-1}\}$, which makes the maximum of $\min_{s \in \{s_1, s_2 \cdots s_{i-1}\}} Dis(s_i, s)$, until the desired number of clusters are set. The distance of state s to cluster c is given like that: $Dis(c, s) = 1/|c| \sum Dis(s_i, s)$, where object $s_i \in c$. After the initial clusters are set, the rest of objects are assigned to the k clusters.

The hierarchical clustering technique is classified into two major types: agglomerative and divisive, in the viewpoint of the clustering process. Agglomerative methods initially assign each objects to one cluster and stepwise merges the pair of clusters with minimum distance. Divisive methods initially assign the population to one cluster and stepwise divide one cluster into two clusters. Either process is continued till the desired number of clusters are generated. Hierarchical clustering produce a hierarchy of clusters which means its clusters embed or include subclusters, while partition clustering produce disjoint clusters. Particularly, hierarchical clustering is much faster than partition clustering in the circumstance of multi-scale clustering.

According to the limitation of test resource and testing requirement, it is feasible to generate any number of clusters, up to the number of states obtained from the candidate test suites. In this work, rather than fixed numbers, fixed fractions of the number of GUI states are used to verify our method. 10%~50%, 5% as the interval, of the states are considered. Particularly, the numbers less than 10% may be too small according to the total number of the population which leads to an inaccurate clustering result. For example, the test cases are classified into two clusters when the test suite is with 100 test cases, and then the defects detected by two test cases would vary a lot. Finally, a subset of test suite needs to be chosen for evaluation against requirement. And many sampling strategies have been proposed for the selecting procedure, such as small cluster sampling, n-per-cluster sampling, adaptive sampling etc. Considering that the goal of clustering is to filter a set of respective test cases, we employ the one-per-cluster sampling strategy in our experiments.

IV. Empirical Study

Having presented the model of GUI state and metric of relationship of GUI states, we now conduct an empirical study to show the rationality.

A. Research Questions

There are two main research questions that will be investigated in this work:

RQ1: Is a subsection of test cases with extensive coverage of GUI states more reliable and efficient for revealing defects in regression testing? The answer to this question will help us to estimate whether the state-coverage-based method is suitable for GUI test case selection. Specifically, we directly compare the state-coverage-based method with the simple random sampling method using two event-driven GUI applications.

RQ2: When dealing with objects with weak transitive relationship, can different cluster approaches get the conforming results, and which approach gets better results? The former answer may help strengthen our confidence of conclusion in RQ1. The later answer will provide a guidance for a better result when applying state-coverage-based method.

B. Subject Applications

We choose two open-source applications written in java with different sizes. The two application are *TerpSpreadSheet* (*TSS*) and *TerpWord* (*TW*), and both are selected from the *TerpOffice3.0 Suite* [11], which was developed by senior software engineering students in the University of Maryland. Table I shows the key information of the two applications. The faulty versions of both the two GUI applications can be downloaded from [12]. Test suites of the two GUI applications each contain n-length test cases generated from *event flow criterion* [3], where n values from 1 to 10. The sizes of the associating test suite are 500 and the detectable defects are 121 and 47.

TABLE I. Key Information of GUI applications

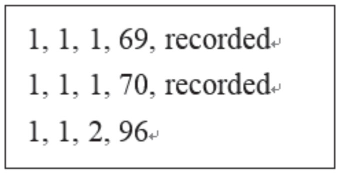
Application	TSS	TW
Windows	9	26
Widgets	176	617
Lines of code	5381	9917
Classes	135	197
Methods	746	1380
Faults Versions	234	295

C. Effectiveness Metrics

In this paper, we are interested in the defect detection effectiveness of a selected subset of existing test suite, that means, the more different defects are revealed, the more effective the subset are. Our experiments had a few basic ingredients: all the defects once triggered can be accepted and recorded, and each detected defect shares the same weight as to effectiveness. Therefore, the total number of the different defects detected by a group of test cases is taken as a metric to estimate the effectiveness of the group.

D. Data Collection

JavaGUIRunner platform developed in java by us, was used in all experiments for data collection. Based on *Java Reflection*, the platform automatically execute each of the test cases stored in the *XML* files against the GUI applications. After a test case was executed, the platform could record two kind of data: the GUI states and the defect detection information. Fig. 3 shows the format of the defect detection information. The first three numbers respectively mean the test suite name, the test case name and the step number. The fourth number represents which defect was detected. The word "recorded" means the defect it follows has been detected by another test case before. Fig. 3 shows the first test case in test suite 1 finds three defects: numbered 69, 70, and 96, and the numbered 96 is the first time to be found.



```
1, 1, 1, 69, recorded↵
1, 1, 1, 70, recorded↵
1, 1, 2, 96↵
```

Figure 3. Defect detection information

E. Experimentation Procedure

We have described the mechanism of state-coverage based method to select a representative subset of test cases for a quick regression testing. The concrete process which includes five main steps are as follows:

Step 1: Execute the entire test suite against fault-free GUI application to generate the GUI state profiles.

In this step, GUI states of all the test cases are recorded by *JavaGUIRuner*, and then the test cases generating the same GUI state are firstly integrated into a cluster. After this step, TSS gets 159 different states, TW gets 217 different states.

Step 2: Clustering procedure is adopted to subdivide the GUI state population into k cluster.

In this step, different cluster approaches: partitioning clustering and hierarchical clustering are respectively adopted. And the k is the final wanted number of clusters, which values from fixed fractions of the number of GUI states.

Step 3: Classify all the test cases according to the cluster of its state.

In this step, we cluster the GUI states to classify the test cases with the expectation that test cases in the same cluster may go through similar run-time states and have similar defect detective ability.

Step 4: Randomly select one test case from each cluster to constitute the seed suite for regression testing.

Our intention is to select a subset of test cases with extensive representativeness, so one-per-cluster strategy is adopted. Then we use the selected test cases to generate the seed suite *XML* files.

Step 5: Run the seed test suites against the faulty versions of GUI applications.

In this step, all the defect are inserted into one edition, and the defect detection information of each test case is record by the *JavaGUIRunner* platform. We repeat the step 4 and 5 100 times to obtain the average detection ability.

F. Results and Analysis

We now present the test case selection results of comparing the state-coverage method with random sampling method. Fig. 4 and 5 respectively show the subset size vs. the number of detected defects for TSS and TW. From the figures, we can see that for both subject applications, all the numbers of defects detected by state-coverage method are much higher than the numbers of defects detected by random sampling method, and the different clustering approaches get conforming results for state-coverage method. That is to say, the state-coverage method is more efficient and reliable than random sampling method, and state-coverage method is quite suitable for GUI test cases selection which answers RQ1. Besides, Fig. 4 shows that in TSS, partitioning clustering approach gets better results for expected numbers which are less than 20%, and similar results for other cases comparing to hierarchical clustering approach. Fig. 5 shows a more stable trend in TW that partitioning clustering approach always get a little better results than hierarchical clustering approach. Therefore, in term of performance, partitioning clustering approach is more suitable than hierarchical clustering approach when dealing with the relationship of GUI states, which answers RQ2.

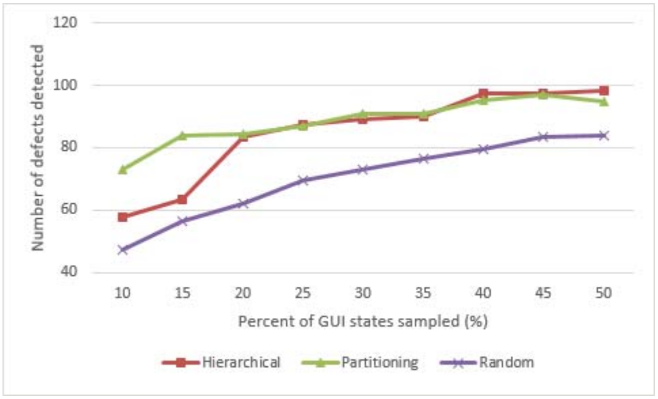


Figure 4. Number of defects detected vs. percent of GUI states sampled of TSS

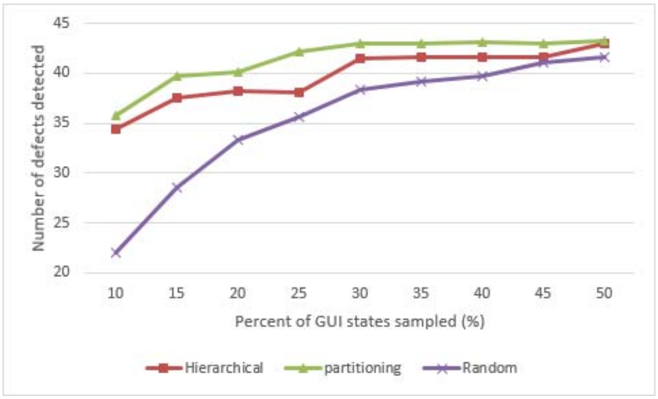


Figure 5. Number of defects detected vs. percent of GUI states sampled of TW

The results generally show that with more clusters to thin GUI states, the selected subset trends to find more defects. One reason may be more test cases are selected, the other reason may be that defects are clustered together and the repetition rate of finding the same defect trends to be reduced. Furthermore, there is also existing an upper limitation, such as when the expected number is more than 40% and 30% of states respectively for TSS and TW, it should be considered whether the cost of classifying states is worthwhile according the amount of the GUI states population. The other notable feature is that the selected subset benefits most from our state-coverage method when the expected number of test cases is less than 20% of the number of GUI states, which suggest the ideal test investment should afford to sample test cases around 20% of the number of GUI states.

It is also important to note that even covering all the GUI states in an execution population may not guarantee to find all the defects, while our state-coverage method is used to improve reliability of estimating a program, and to increase the efficiency of regression testing by potentially clustering the test cases related the same defect into one cluster.

In addition, as the test cases related with the same defects potentially cluster together, it is easier to find several different trigger cases to help rapidly obtain the data and boundary condition which are causing the same problem in debugging procedure. The distribution of the clusters may provide an indication of how often and serious a defect affects users. For example, a test case in the big-size clusters potentially have more similar test cases that trigger the same defect, so the defects consisting in big-size clusters are more likely to encumber users.

G. Threats to Validity

Where external validity is concerned, we have only applied our method to two GUI applications which are constructed in a similar manner, thus the small sample size may not be representative of the broader population of programs. Experiments on multiple programs of different size and manner will strengthen the validity. Additionally, for the difference in the underlying goals, we have just compared our method with a baseline approach, but not considered other approaches. Universal standard for evaluating different approaches must be investigated in further study.

Where internal validity is concerned, the faults in our tools which are used to simulate our regression testing procedure may cause problems. We use *JavaGUIRunner* platform and many other programs to implement our experiments. To control the threat, plenty of test has been conducted during the development of the tools.

V. Related Work

Although the test cases selection for traditional software has been deeply investigated, little research has focused on GUI applications. The main literature is the work by Leung and White, who applied the Firewall approach to GUIs [13]. They categorized modules into different categories, particularly the call pairs between unchanged module and changed module forming the so-called firewall. Then the test cases were selected based on Complete Interactions Sequences (CIS) in order to select all the modification-traversing test cases. Leung and White maintained a very conservative approach, but they also noted that the Firewall approach is not very reliable for there is always a risk that a fault-revealing test case is outside the given test suite.

There is a close relationship between test suite minimization and test case selection, for both are about choosing a subset of test cases from test suite. The key difference between the two techniques is the minimization techniques permanently eliminate the redundant test cases while selection techniques temporarily select a subset of test cases. McMaster and Memon proposed a test suite minimization technique for GUI applications based on call stack coverage [14]. They represented a test suite by a set of unique maximum depth call stacks, and then minimized the test suite to a subset with same set of unique maximum depth call stacks. Another similar technique, to aid the GUI regression testing, is test case prioritization. Prioritization technique seeks an ideal ordering of test cases for early fault detection, which can be prematurely halted after the first k tests being conducted. Bryce and Memon applied the principles of interaction coverage to GUI test case prioritization [15]. Empirical study shows that interaction coverage is more efficient than event coverage, longest to shortest, and shortest to longest for test case prioritization.

In fact, there is no effective procedure which can always help to find fault revealing test cases, however, under certain conditions regression testing can be more reliable. In this work, we restrict our attention on GUI corrective regression test case selection [16] which does not invoke changes in specifications, and we further restrict our attention on state-based test case selection, which relies on the analysis of state of P to select test cases for P'.

VI. Conclusion

GUI applications are quite different from conventional software which severely need systematic and scalable regression testing. In this paper, we present a state-coverage based method that provides manageable number of test cases for GUI regression testing. This paper proposes a novel GUI state model to address the execution of test cases, and apply automatic cluster analysis to divide the population of GUI states into several clusters so as to classify the test suite. One-per-cluster strategy is adopted to balance the representativeness with redundancy. Taking GUI state as a substitute of test case helps to classify the suite into several subsets, so that test cases in each subset potentially execute similar code and detect similar defects. Two experiments on GUI applications are conducted and the results demonstrate that state-coverage based method is feasible and efficient to sample test cases for GUI applications. Moreover, in term of performance, partitioning clustering approach is better than hierarchical clustering approach at dealing with GUI states.

This work is only a preliminary study on state-coverage method, and plenty of work needs to be further investigated. Firstly, although we were able to apply our method to analysis the GUI states for the TerpOffice applications, the price to analysis the data for the larger applications may be expensive. Thus, we intend to further refine our GUI state model to reduce the cost and maintain the accuracy. One idea to simplify our GUI model, is to get rid of the redundant widgets whose properties nearly never change. Another strategy is to consider a weight property for each window operational profile according to its complexity to increase the accuracy. Furthermore, more empirical studies on different kind of GUIs, such as Web applications, should be done to strengthen the applicability. Finally we want to explore GUI test case prioritization based on our cluster analysis of GUI states.

REFERENCES

[1] A. M. Memon and M. L. Soffa, Regression testing of GUIs[J]. Acm Sigsoft Software Engineering Notes, 2010, 28(5):118-127.

[2] G. Rothermel and M. J. Harrold A framework for evaluating regression test selection techniques[C]// Proceedings - International Conference on Software Engineering. 1994:201-210.

[3] A. M. Memon, M. L. Soffa, and M. E.Pollack, Coverage criteria for GUI testing[C]//ACM SIGSOFT Software Engineering Notes. ACM, 2001, 26(5): 256-267.

[4] B. N. Nguyen, and A. M. Memon, An observe-model-exercise* paradigm to test event-driven systems with undetermined input spaces[J]. Software Engineering, IEEE Transactions on, 2014, 40(3): 216-234.

[5] Z. B. Gao, C. Fang, and A. M. Memon, Pushing the limits on automation in GUI regression testing[C]//Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on. IEEE, 2015: 565-575.

[6] Q, Xie, and A. M. Memon, Designing and comparing automated test oracles for GUI-based software applications[J]. ACM Transactions on Software Engineering and Methodology (TOSEM), 2007, 16(1): 4.

[7] D. Leon, and A. Podgurski, A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases[C]//Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on. IEEE, 2003: 442-453.

[8] S. Yoo, M. Harman, P. Tonella, et al. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge[C]//Proceedings of the eighteenth international symposium on Software testing and analysis. ACM, 2009: 201-212.

[9] J. Feng, B. B Yin., and K. Y. Cai, et al. 3-Way GUI Test Cases Generation Based on Event-Wise Partitioning[C]//Quality Software (QSIC), 2012 12th International Conference on. IEEE, 2012: 89-97.

[10] K. Y.Cai, L. Zhao, H. Hu, et al. On the test case definition for GUI testing[C]//Quality Software, 2005.(QSIC 2005). Fifth International Conference on. IEEE, 2005: 19-26.

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

[12] A. M. Memon, http://www.cs.umd.edu/~atif/TerpOfficeWeb/TerpOfficeV3.0/index.html, 2010.

[13] L. White, H. Almezen, and S. Sastry, Firewall regression testing of GUI sequences and their interactions[C]//Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on. IEEE, 2003: 398-409.

[14] S. McMaster, and A. M. Memon, Call-stack coverage for gui test suite reduction[J]. Software Engineering, IEEE Transactions on, 2008, 34(1): 99-115.

[15] R. C. Bryce, and A. M. Memon, Test suite prioritization by interaction coverage[C]//Workshop on Domain specific approaches to software test automation: in conjunction with the 6th ESEC/FSE joint meeting. ACM, 2007: 1-7.

[16] S. Yoo, and M. Harman, Regression testing minimization, selection and prioritization: a survey[J]. Software Testing, Verification and Reliability, 2012, 22(2): 67-120.