ISBN 978-981-11-3671-9
Proceedings of 2017 the 7th International Workshop on Computer Science and Engineering (WCSE 2017)
Beijing, 25-27 June, 2017, pp. 188-194
doi: 10.18178/wcse.2017.06.033

GUI Test Suite Reduction Based on Running Information

Juncheng Chen [1 +] and Hua Wu [2]

[1] BJUT Faculty of Information Technology, Beijing University of Technology Beijing, 100124, China
[2] Network and Educational Technology, Anyang Normal University, Anyang, 455000, China

Abstract. Running GUI (Graphic User Interface) test cases is a time-consuming task. In order to improve test effectiveness, it is necessary to eliminate redundant test cases from test suite. This paper presents an approach for reducing test suite based on test cases' running information, which includes path information of event handler function and invoked method sequence of running threads. In this approach, a relationship between two test cases is defined, and some test cases are regarded to be redundant according to the relationship and the running information. Based on the idea, we implement a prototype for Windows GUI applications. Initial experiment shows that the test suite produced by our approach has larger size and higher fault detection effectiveness than that produced by call stack coverage approach[1], and running reduced test suite save about 19% time compared to the original.

Keywords: GUI testing, test suite, reduction, running information

1. Introduction

In GUI testing, running GUI test cases is a very time-consuming task [2]. Size of test suite may be increased with the progress of software development, which may lead to incremental costs. A direct approach to lower test cost is reducing redundant test cases.

A lot of researchers have focused on the test suite minimization. Harrold et al. proposed a technique to select a representative set of test cases from a test suite that provides the same coverage as the entire test suite [3]. Some techniques reduce test suite through selecting test cases which satisfy more requirements [4]. In order to improve fault detection effectiveness, Jeffrey et al. presented a test suite reduction technique, which use two test coverage criteria to decide whether a test case is redundant [5]. Smith et al. presented a test suite reduction technique based on the call tree path [6]. Considering various test coverage criteria and fault detection effectiveness, Hwa-You Hsu et al. put forward a general framework for test suite reduction, this framework provides methods to encode a wide range of test-suite, support multi test coverage criteria and give reduction solutions [7]. In GUI implementations, however, codes called via an event handler may be executed in many different contexts, which may uncover faults, due to the increased degree of freedom that modern GUIs provide to users, which makes conventional reduction techniques difficult be applicable to GUI applications [1].

Running a GUI test case spends more time than a non-GUI test case, because the GUI test case must locate UI controls, mock user's behaviors on UI controls and check its correctness from user's perspective, which are very costly. It is difficult to shorten the execution-time for GUI test cases. So, reduction of GUI test suite is direct and meaningful for GUI testing.

Scott et.al proposed a call stack coverage for GUI test suite reduction [1]. The approach defined maximum depth call stack for a thread, denoted as $C(t)$max, and collected all maximum depth call stack for all threads in a running test case tc, denoted as $Cmax(tc)$. If $Cmax(\mathrm{tc}_1) = Cmax(\mathrm{tc}_2)$, the authors regard tc_1 or

[+] Corresponding author. Tel.: +86-010-67392370-605
E-mail address: juncheng@bjut.edu.cn.

tc_2 as a redundant test case. Empirical studies showed that call stack coverage-based test suite reduction produces better results for GUI applications compared to traditional techniques.

```
1 private void comboStatus DrawItem ( … ) {
2  …
3  switch ( b e w s t a t u s ){
4     case Online :
5       brush =Brush . Green ;
6       break ;
7     case Busy :
8       brush =Brush . Red ;
9       break ;
10    . . .
11 }
12 . . .
13 e . Graphic . F i l l R e c t a n g l e ( brush , . . . ) ;
14 . . .
15 }
```

Fig. 1: Code snippet of example.

However, only considering the call stacks may result in deleting non-redundant test cases and losing starting order of the running threads. For example, code snippet in a GUI application depicted as in Fig 1 may lead to deleting a non-redundant test case.

Assumes tc_1, tc_2 execute statement in line 5-6, line 8-9, respectively, and other executing statements in tc_1 is same to those in tc_2. tc_1 and tc_2 can't be regarded as equivalence because they execute different paths. However, tc_1 and tc_2 are regarded as equivalence and one of them will be deleted in [1].

In a multi-thread GUI application, there usually is a main thread, which is a UI thread in most situations, and other threads started by methods in the UI thread in most situations. So the starting order of each thread is very critical to decide the test case redundance.

To avoid the two shortcomings, we define a relationship between two test cases and propose a new GUI test suite reduction approach based on running information, which includes execution path in event handler functions and invoked method sequence of running threads, of test case. Some test cases are regarded as redundance based on the relationship and the running information.

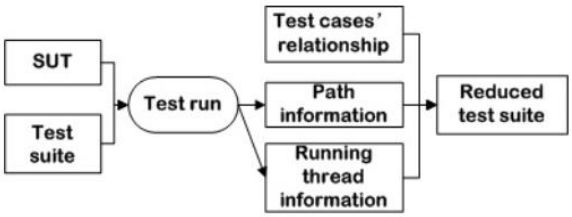

Fig. 2: Test reduction process based on running information.

The rest of paper is organized as follows. Section 2.1 defines contains relationship between two test cases, and puts forward a new idea of reducing test cases, section 2.2 explains the implementation technique for Windows GUI applications, section 2.3 shows an initial experiment and discusses the result. At last, we give conclusion in section 3.

2. Our Approach

2.1. Our Idea

In our approach, reduced test suite is produced as Fig 2. Firstly, a relationship between test cases is defined; Secondly, run test cases and collect path information in event handler functions and running threads information; Thirdly, remove redundant test cases based on the relationship and the path information and running threads information.

A GUI test case tc is represented by the form $< S_0,f_1,f_2,\cdots,f_n,$ exit $>$ $(n \geq 1)$, where S_0 represents the initial GUI state[2] when SUT(System Under Test) is loaded, $f_i(1 \leq i \leq n)$ represents a GUI event handler function[8], and exit represents an event handler function which close the SUT. Event handler functions implement functionalities of the SUT and their implementations decide SUT's quality. For convenience of description, $t_{ci\text{-}j} =< f_i,f_{i+1},\ldots,f_j > (1 \leq i \leq j \leq n)$ is used to represent a test case's subsequence from function fi to function f_j in t_c.

Based on the definition of GUI test case, *Contains* relationship between two test cases is defined as the following.

Contains: Given two test cases

$$t_1 =< S_0,f_1,f_2,\ldots,f_{k1};f_{k2};:::;fn,exit > \text{ and}$$
$$t_2 =< S_0,f_1,f_2,\ldots,f_{k1},f_{j1},\ldots,ffj_2,f_{k2},\ldots,fn,exit >,$$
$$(1 \leq k_1 \leq k_2 \leq j_1 \leq j_2 \leq n)$$

if $t_{c11\text{-}k1},t_{c1k2\text{-}n}$ are identical to $t_{c21\text{-}k1},t_{c2k2\text{-}n}$, respectively, then t_2 contains t_1, denoted as $t_1 \sqsubseteq t_2$.

if $t_1 \sqsubseteq t_2$, and the running information of $t_{c11\text{-}k1},t_{c1k2\text{-}n}$ is identical to those of $t_{c21\text{-}k1},t_{c2k2\text{-}n}$, then we consider t_1 is redundant and can be eliminated from test suite.

Strictly speaking, running information should include related complete execution path and related complete context information. However, getting and analyzing the running information may be very costly. For GUI test case's subsequence t_{cs1}, t_{cs2} of t_{c1}, t_{c2}, respectively, if related execution path in GUI event handler functions and the invoked method sequence in related threads of t_{cs1} is identical to those of t_{cs2}, then t_{cs1} and t_{cs2} are regarded as to be equivalent.

In order to describe the equivalence relation between two test case's subsequences, related execution path in GUI event handler functions and related invoked method sequence in threads must be captured.

In a GUI application, there are usually two kinds of class, GUI window class and non-window class. GUI window class defines GUI elements and implements event handler functions, and non-window class usually defines internal objects and implements underlying complicated logic computation.

Execution path in a GUI event handler function *ehf* is a path in ehf's ICFG(Inter-procedural Control Fow Graph) which is constructed as [9]. In ICFG of *ehf*, the following invocations are regarded as inter-procedural call.

- ehf invokes function defined in window class;
- ehf invokes static function;
- function defined in window class invokes function in window class;
- function defined in window class invokes static function.

Given two test cases' subsequence t_{cs1} and t_{cs2}, $patht_{cs1}$, $patht_{cs2}$ represents execution path in event handler functions of t_{cs1}, t_{cs2}, respectively. And $patht_{cs1} = patht_{cs2}$, $patht_{cs1} \neq patht_{cs2}$ represent situations where $patht_{cs1}$ is and isn't identical $patht_{cs2}$, respectively. Obviously, if $patht_{cs1} \neq patht_{cs2}$, then t_{cs1} is not equivalent to t_{cs2}. Even if $patht_{cs1} = patht_{cs2}$, t_{cs1} and t_{cs2} can't be regarded as equivalence because the execution path in GUI event handler functions only represents user's abstract behaviour, not refers to the underlying execution information.

Underlying execution information can be represented by invoked method sequence of running threads. For each running GUI application, there may be several running threads. Among those threads, the UI thread usually is the main thread, and other threads are started by methods in the UI thread. In this paper, we only consider the UI thread and threads started by methods in UI thread, because GUI testing focus on the correctness of event handler functions, codes of which are running in the UI thread or the threads started by methods.

Given a GUI test case $t_c =< S_0,f_1,f_2,\ldots,f_n;exit >$, t_c's UI thread and other related threads started by methods in UI thread is showed as Fig 3. In the figure, the middle stack represents invoked method sequence of UI thread, other four stacks represent invoked method sequence of threads started by method in UI thread. In each stack, methods are invoked one by one from the bottom to the top of the stack , and the method at stack bottom and top represent the first and the last invoked method in the thread, respectively. Init and exit in UI thread indicates initial and terminal activities which contains a series of invoked method.

In Fig 3, concerning threads are essentially organized into a tree, of which each node represents a method call. UI thread is the trunk of the tree, and other threads represent branches of the tree. Underlying running information of a test case t_c can be represented by a tree , denoted as UIT_{tc}, and we call it *running UI thread tree.*

Assumes t_c =< S_0,f_1;f_2,...,f_{k1};f_{k2},...,f_n,*exit* > and its running UI thread tree UIT_{tc}, t_{cs} = $t_{c1\text{-}k1}$ is a subsequence of tc. Invoked method sequence of tcs is represented by a subtree of UIT_{tc}, called *running UI sub-tree*, in which the trunk of the tree UI_{sub} =< *Init*,f_1;m_{11};m_{12},...,f_{k1},m_{k11},m_{k12},…,m_{k1m} >(in UI thread, method m_{k1m}'s successor is f_{k2}), and the branches are the threads invoked by methods in UI_{sub}. UIT_{tcs} is used to represent the related running information for test case's subsequence *tcs.*

Assumes t_1 =< S_0,f_1,f_2,…,f_{k1},f_{k2},…,f_n,exit > and t_2 =< S_0,f_1,f_2,…,f_{k1},f_{j1},…,f_{j2}, f_{k2},…,f_n;exit > ($1 \le k_1 \le k_2 \le j_1 \le j_2 \le n$), t_{cs11} = $t_{c11\text{-}k1}$;t_{cs12} = $t_{c1k2\text{-}n}$, t_{c21} = $t_{c21\text{-}k1}$, t_{c22} = $t_{c2k2\text{-}n}$. if $t_1 \sqsubseteq t_2$, and t_{c11},t_{c12} are identical to t_{c21}, t_{c22}, respectively, then t_1 is redundant if and only if the following condition is satisfied.

$$pathtc_{s11} = patht_{cs21} \cap patht_{cs12} = patht_{cs22} \cap UIT_{tcs11} = UITt_{cs21} \cap UITt_{cs12} = UITt_{cs22}$$

2.2. Implementation

To implement our approach, all the test case pairs, in which one test case contains another, are selected firstly. Secondly, we run all test cases and collect execution path information and running threads information. Thirdly, we collect and analyze execution path information and running threads information, and produce a reduced test suite according to the rule described in section II.

Among above steps, the second step and the third step are critical to our approach. In the second step, in order to get the test case execution path information, we scan source code of functions in ICFG described as as section II. Whenever there is a branch, a monitoring statement will be inserted. When the branch is executed, the monitoring statement will notify related path information. The number of branches in considering functions is limited and the process of instrumenting monitoring statement can be finished in $O(N)$, where N is total source code scale of considering functions.

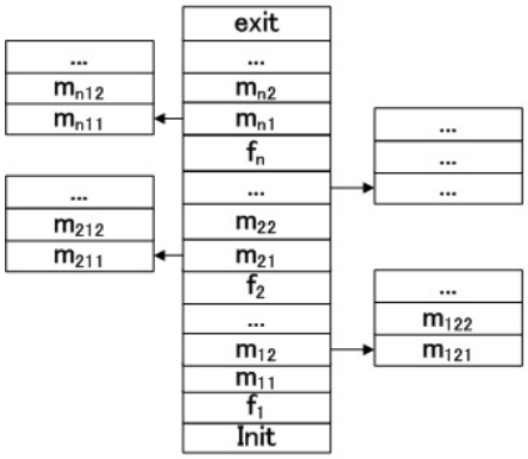

Fig. 3: Running UI thread.

In the third step, for Microsoft compiler environment, a function named *_penter* will be invoked whenever any method is invoked. We re-implement this function and use the form (*Id,ThreadId,MethodName*) to record the invoked method, where Id represents the order in invoked method sequence, the first invoked method's *Id* is 1, and the second is 2, and so on, ThreadId represents unique thread number, and MethodName is the invoked method's name and it runs in the thread whose thread number is *ThreadId*.

Running a test case produces path information in event handler functions and an invoked method sequence MS, each node in the method sequence has the form (*Id,ThreadId,MethodName*). Getting and comparing path information of test case's subsequence can be finished easily. If related path information of two test cases' subsequence are not identical, there is no redundance for the two test cases, otherwise, we must get and compare related running UI subtrees of the test cases' subsequence. In order to get the running

UI sub-tree, the running UI thread tree should be constructed at the first. Algorithm 1 describes how to transfer an invoked method sequence into a running UI thread tree.

Algorithm 1 transfer a method sequence to a running UI thread tree

```
Input:
    FS;
Procedure:
  1: UIThreadId = GetUIThreadId(FS);
  2: UIThread = GetUIThread(FS;UIThreadId);
  3: SetChunk(UITree;UIThread);
  4: RelatedThreads = GetRelatedThreads();
  5: for all thread ∈ RelatedThreads do
  6:   FirstId = GetFirstFunc(thread);
  7:   MaxId = GetMaxLessId(UIThreads;FirstId);
  8:   SetBranch(UITree;MaxId;thread);
  9: end for
Output:
    UITree;
```

In Algorithm 1, Line 1 gets UI thread Id which represents main thread in the SUT, Line 2 gets a method sequence in the UI thread, and Line 3 sets the UI thread in UITree and makes it a chunk. Line 4 gets all the threads. For each thread, Line 6 gets the first invoked method's order *FirstId* in *MS*, Line 7 look for the max number of order *MaxId*, which is less than FirstId, in the main thread, and Line 8 makes the thread a branch, whose first invoked method is a successor of the method whose order is *MaxId*.

In Algorithm 1, time complexity of getting all the threads is $O(N)$, where N is the number of nodes in function sequence, and time complexity of constructing a running UI thread tree is $O(kM)$, where k is the number of threads and M is the number of functions in main thread, in worst case.

The rest task is to analyze running UI thread sub-tree of the related test cases' subsequence. Given a test case t_c =< S_0,f_1,f_2,…,f_n,*exit* >each function in t_c, such as f_1 and f_2, appears in the function sequence of main thread. Map between functions in tc and functions in invoked method sequence of the main thread can be constructed in $O(M)$, where M represents number of the invoked method sequence in the main thread. According to the map, we get the related running UI thread sub-tree conveniently. So, comparing and analyzing the running UI thread sub-tree of the related test cases' subsequence is a simple task

2.3. Experiments

In order to verify our approach's effectiveness, we implement a prototype tool in win32 environment. Through inputting a series of abstract GUI test cases, each of which has the form < *S0,f1,f2, …, fn, exit* >, and running information of test suite into the tool, we produce a reduced test suite.

The experiment is carried out in windows environment, and we select an open source application named Miranda *IM*[10] as our SUT. This application is a instance message tool which support many popular protocols such as *MSN*, *ICQ*, *IRC*, *Jabber*, and so on, and it is widely used as a client software.

In the experiment, we generate EFG model using the algorithm in Guitar, and generate test cases according to the eventinteraction coverage criteria based on the EFG model[11], and get 427 test cases, among which 39 test cases are invalid. The table I compares our approach(represented by pr) with call stack coverage approach(represented by *cs*) in size reduction and faults detected of reduced test suite. *FD* represents fault detection number.

TABLE 1: Comparing Data Between our Approach and Call Stack Coverage Approach

Original		cs		pr	
size	FD	size	FD	size	FD
388	26	268	15	294	17
% Reduction from original		30.09	42.31	24.43	34.62

The table I shows that the reduced test suite containing 294 and 268 test cases are produced using our approach and cs, respectively, which indicates that our approach is more preserve than cs. In order to verify ability of finding bugs of our approach, we manually inject 40 errors in the SUT's source code and generate 40 mutations for the SUT, and each mutation inject one error. The result shows that reduced test suite produced by our approach kill 17 mutations, that by *cs* kill 15 mutations, which indicates that test suite produced by our approach has larger size and higher fault detection effectiveness than that by call stack coverage approach.

In the experiment, time is spend on the following task:

- It takes us about 3 hours to get the relationship between the user behavior in test case and event handler function in source code.
- It takes about 10.2 hours and 8.5 hours to run 388 test cases with and without collecting running information, respectively, which indicates that the process of collecting running information spends about 1.7 hours. After eliminating the redundant, it takes about 6.9 hours to run the reduced test suite without collecting running information. This shows that running reduced test suite save 1.6 hours, which is about 19% of running original test suite.
- Analyzing the collected information spends about 20 minutes.

The result shows that the reduced test suite has larger size and higher fault detection ability using our approach than that using call stack coverage approach in the case, and about 19% time is saved when running reduced test suite for the SUT, which indicates that additional effort for test reduction is worth-while, especially for regression testing

3. Conclusion

In this paper, we define a relationship between two test cases, and present a new GUI test reduction approach, which avoids two shortcomings in call stack coverage approach. In this approach, we collect execution path information in event handler function and running UI thread information of test cases, analyze the information and eliminate redundant test cases based on the analysis result. Initial experiment is carried out on Windows environment, the result shows that about 24% test cases are regarded as redundant and eliminated and about 19% time is saved in running reduced test suite in the case. Comparing to call stack coverage approach, test suite produced by our approach has larger size and higher fault detection effectiveness.

4. Acknowledgements

The authors thank the anonymous reviewers for their feedback which helped improve this paper. This work has been partially supported by the Beijing Postdoctoral Research Foundation.

5. References

[1] S. McMaster and A. Memon, "Call-stack coverage for gui test suite reduction," *Software Engineering, IEEE Transactions on*, vol. 34, no. 1, pp. 99 –115, jan.-feb. 2008.

[2] X. Yuan, "Feedback-directed model-based gui test case generation," Ph.D. dissertation, University of Maryland, 2008.

[3] M. J. Harrold, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," *ACM Trans. Softw. Eng. Methodol., vol*. 2, no. 3, pp. 270–285, Jul. 1993. [Online]. Available: http://doi.acm.org/10.1145/152388.152391

[4] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *Proceedings of the 2007 international symposium on Software testing and analysis, ser.* ISSTA '07. New York, NY, USA: ACM, 2007, pp. 140–150. [Online]. Available: http://doi.acm.org/10.1145/1273463.1273483

[5] D. Jeffrey and N. Gupta, "Improving fault detection capability by selectively retaining test cases during test suite reduction," *Software Engineering, IEEE Transactions on*, vol. 33, no. 2, pp. 108 –123, feb. 2007.

[6] A. M. Smith, J. Geiger, G. M. Kapfhammer, and M. L. Soffa, "Test suite reduction and prioritization with call trees," in *Proceedings of the twentysecond IEEE/ACM international conference on Automated software*

engineering, ser. ASE ’07. New York, NY, USA: ACM, 2007, pp. 539– 540. [Online]. Available: http://doi.acm.org/10.1145/1321631.1321733

[7] H.-Y. Hsu and A. Orso, “Mints: A general framework and tool for supporting test-suite minimization,” in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, may 2009, pp. 419 –429.

[8] L. Zhao and K.-Y. Cai, “Event handler-based coverage for gui testing,” in Quality Software (QSIC), 2010 10th International Conference on, july 2010, pp. 326 –331.

[9] H. Pande, W. Landi, and B. Ryder, “Interprocedural def-use associations for c systems with single level pointers,” *Software Engineering, IEEE Transactions on, vol*. 20, no. 5, pp. 385 –403, may 1994.

[10] miranda, “mirand,” 2012. [Online]. Available: https://code.google.com/p/miranda

[11] A. M. Memon, “A comprehensive framework for testing graphical user interfaces,” Ph.D. dissertation, University of Pittsburgh, 2001.

[12] Dharmender Singh Kushwaha Avinash Gupta, Namita Mishra. Rule based test case reduction technique using decision table. In 2014 IEEE International Conference on Advance Computing, Pages 1398-1405, Los Alamitos, 2014. IEEE Computer Society.