Asian Journal of Information Technology 15 (20): 4048-4053, 2016
ISSN: 1682-3915
© Medwell Journals, 2016

Test Suite Reduction (TSR) Recommendation in Function Calls Using Sliding Window

[1]C.P. Indumathi and [2]K. Selvamani
[1]Department of CSE, Anna University, Trichy, India
[2]Department of CSE, Anna University, Chennai, India

Abstract: The software modularity makes immense importance in the process of software development life cycle as all the software's are developed by COTS (Commercial Off The Shelf) components which are mainly concerned with size and complexity. This introduces challenges to existing test suite reduction techniques and makes it unsuitable for readymade software and for large complex software's. In this study we propose(recommend) a method using sliding window in function calls to achieve better test suite reduction, without accessing the source code and also to reduce the cost of integration testing by taking size, cost and time as motivating factors. Our proposed method provides good result with earliest reduction techniques in literature.

Key words: Function call, integration testing, sliding window, software testing, test case reduction

INTRODUCTION

In this modern era, millions of software's were released every month. Among them, most of the softwares are highly modular in nature as well as integration testing plays a major strength in the process of software development. Also, a successive evolution of software releases with large number of modifications increases the cost for regression testing (He *et al*., 2005). Hence, test-suite reduction technique (Delamaro *et al*., 2001; McMaster and Memon, 2005; Rothermel *et al*., 2002) is essential to decrease the overall cost of this regression integration testing. Various traditional approaches such as mutation testing, code coverage criteria (Harrold *et al*., 1993; Haley and Zweben, 1984) depend upon the source code of interacting module to reduce the size of original test-suite. Its source code depending strategy makes them unsuitable for software whose source code size is fairly large and for COTS (Commercial Off-The-Shelf) components which is normally supplied without source code. Hence, this research study aims in proposing a method to reduce the test suite size without accessing source code based on the sequence of function calls between the modules.

Existing approaches: Conventional approaches to test suite reduction was established on the various coverage criteria and program elements which is divided into two broad groups namely:

- Interface mutation analysis
- Structural metrics

Interface mutation analysis: Interface mutation (Piwowarski *et al*., 1993; Kichigin, 2007) method relies on the three categories to estimate the completeness of integration testing namely:

- Operators that simulate integration errors are used as mutation operators
- To test more than two modules, the interaction between several modules has been tested pair-wise
- Mutation operators are practically applied to such "interface" parts of the source code as calls to interface functions their parameters or global variables

Interface mutation operates with source code which makes it unsuitable for applications. For e.g., considering for any application it must focus on the parameters such as:

- Interface and links between applications
- Interface to related systems or applications
- Features, functions and facilities

Integration testing type considers on testing the interfaces (alone). It is one of the subset of the integration testing phase whereas Integration testing phase focuses on revealing defects when combining various components including usage, incomplete understanding of product domain, user errors and so on. Hence, the integration testing focuses on both interface and usage flow.

Corresponding Author: C.P. Indumathi, Department of CSE, Anna University, Trichy, India

Structural metrics: Structural metrics are based on testing the behavior of structural elements of the software which are responsible for interaction of elements being integrated. Structural metrics are calculated in two stages.

Static stage: At this stage, the source code of modules to be integrated is analyzed to reveal dependencies between them.

Dynamic stage: At this stage, the interaction between modules during the software execution on a test suite is analyzed to check whether the dependences between the modules revealed at the static stage are actually fulfilled.

Based on the result of checking, the coverage of interaction between the modules by test suite is estimated. For this, two different groups of metrics are used. One such metric uses program control flow model (Zhu *et al*., 1997; Delamaro *et al*., 2001) whereas other metric uses program data flow model (Delamaro *et al*., 2001; Hofmeyr *et al*., 1998; Harrold *et al*., 1993).

The conventional approaches for integration testing clearly relies on the source code thus making a difficult to use these methods for testing large scale Intensive software and COTS (Commercial Off-The-Shelf) which is a ready-made software components.

Module interaction and integration error: Integration testing plays vital role in software development life cycle. Many components are integrated and tested together which is a daunting task in enterprise applications. Moreover, these diverse teams build different modules and components at the time of module development. There is a possibility of chances by the clients/customers to change in requirements between these new requirements may not be unit tested. Integration testing becomes necessary to analyzes and reveal the differences between the parameters and the required parameters during interaction between components of software. Such differences are termed as integration errors (Rountev *et al*., 2005; Linnenkugel and Müllerburg, 1990). In this research work, the interactions between software modules are performed through the interface of functions. Integration errors are raised when an incorrect value (s) is transferred between interacting modules. Integration errors of four types and are performed as follows.

For e.g., consider the program P1 and test tt for P1. Suppose, if the program include the modules X and Y. Such that X contains calls to Y then,

Input

In (Y): Represents the set of values that is passed from module X to module Y. There are "i" tuples of input values in a call to a function is determined by which the input parameters used in the function call and the global variables used in Y

Output

Out(Y): Represents the set of values that is passed from module Y to module X. The i tuples of input values in a call to a function B is determined by the output parameters used in the function call and the global variables used in Y and the values returned by Y.

There are four types of errors namely Type 1 Error, Type 2 Error, Type 3 Error, Type 4 Error. These errors will cause erroneous outputs based on their input values to the process From the above, three types of Integration Errors were noticed that location of the state is not specified which responsible causes incorrect outputs. Hence, considering existing incorrect values. In Type 4 error, when IN(Y) has the expected values, a fault in Y produces an erroneous output before returning from Y. In this case, there is no error propagation through the connection X-Y. This type of expected error is not detected during unit testing.

Hence, Type 4 error can be ignored since it has no effect on integration testing therefore, the other three types of error are taken into consideration. These three types of integration errors can further form hybrid combinations. The possible hybrid types of integration errors are described as follows.

Type3_Type1 Error: When a module say X sends correct values of IN(Y) to module Y and Y incorrectly computes OUT(Y) which is part of IN (Z) which will cause erroneous output in module Z.

Type3_Type2 Error: When module X sends correct values of IN(Y) to module Y which incorrectly computes that OUT(Y) is part of IN(Z) and module Z returns incorrect output.

Type2_Type1 Error: When module X sends incorrect values of IN(Y) to module Y which leads to incorrect values of that OUT(Y) is part of IN(Z) and this will cause erroneous output at Z.

Type2_Type2 Error: It is similar to Type2_Type1 error with the exception that the module Z will return incorrect values.

MATERIALS AND METHODS

Implementation details: This method is implemented based which is based on the construction models of module interaction dependencies on a test suite. This method does not impose the source code to be instrumented and hence, there is no need to the access the source code of the software.

Model for module interaction: This model of module interaction behavior is constructed using sequences of calls of the module interface functions performed during the execution of programs. The function names and parameters passed to those functions plays a significant role in extracting the interaction behaviors.

Interaction between two testing modules: The following function calls are essential for constructing sequences of calls of interacting modules. We assume that two modules (X and Y) interact through the functional interface of module B. By interface function, we mean the functions that are included into the program interface is a software module.

During the trace of interfacing modules X and Y on test tt, the sequence of the interface functions of Y called by X when executed on the input data to the test case tt. We assume that the functions in the trace are arranged in the order of their calls and for each function, the actual parameters passed in the function call are indicated. When the sequence of interface functions of length K, we call any continuous sequence of length K found in the interaction trace.

By the set of sequences of interface functions (Hofmeyr *et al*., 1998) of length K corresponding to the interaction between modules X and Y on the test tt, we mean that the set of all feasible sequences of length K found in the trace of interaction between modules X and Y on the test tt.

To obtain the set of sequences of interface functions of a fixed length, the moving window technique of the size of K is used (the window size corresponds to the selected length of sequences as shown in Fig. 1).

According to this approach, the sequences are selected in the following way: the first sequence of the trace is selected to be the first K interface functions in a row starting from the first function in the trace; the second sequence is selected to be the interface functions in a row starting from the second function and so on until the entire trace is passed. Model of the interaction between modules A and B on test t is taken to be the set of sequences defined above.

Test suite reduction method: The test suite reduction method allows set of sequences of calls to interface module obtained from the constructed model and given as input to the algorithm proposed. Let us consider TS is the initial test suite, TS' is the reduced test suite such that TS' is a subset of TS(TS' _ TS), tt is the next test from initial test suite (tt _ TS):

$$\text{Mts} \rightarrow \text{interaction behavior of model on test tt}$$

$$\text{MTS}' \rightarrow \text{interaction behavior of set of models on tests of TS}'.$$

The implemented test suite reduction method uses the interaction model described above. Let TS be initial test suite, TS' be the reduced test suite(TS'<-TS), tt be next test from initial test suite(ttETS), Mts is the interaction behavior of model on test tt and MTs' is the interaction behavior of set of models on tests of TS'.

- At first, the subsequent test cases to be tested are selected from the test suite to be reduced and it is executed by program
- The interaction behavior between modules X and Y in the model Mts , test tt is built
- If the model is empty go to step no 1
- If the model is not empty, check whether the model Mts belongs to set of models MTs'
- If Mts does not belong to the set of models MTs', then Mts is added to the set MTs' and the test tt is added to the reduced test suite TS'
- Otherwise if MtS belongs to set of models MTS', then do nothing and neglect tt
- If TS has not been passed, go to step 1 otherwise the reduced test suite TS' is assumed to be constructed and ends the process

It should be noted that, although our study considers only the real number parameters of functions, the proposed approach can be adapted to case of arbitrary parameters of functions for which the similarity relation can be defined .

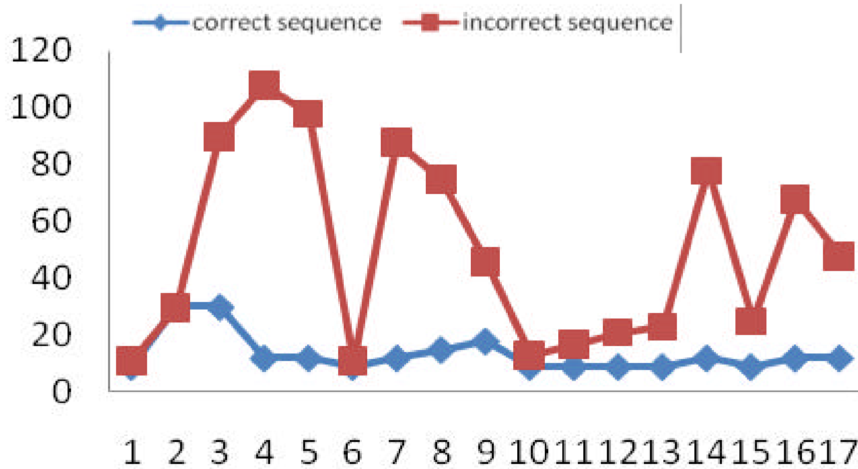

Fig.1: Stack example with sliding window (size = 3)

RESULTS AND DISCUSSION

Experimental results: In this study, we use the test suite reduction method from the implementation details to explore in simple stack application and considered for interaction between sequence of calls. The methods and respective signatures used in the stack application is depicted in Table 1.

Interaction sequence in which stack based application module calls stack module has been extracted and possible combinations of permutations are obtained by fixing a range of sliding window size to find an optimum sliding window size. Then test suite is reduced based on the obtained correct and incorrect sequence of calls for the optimum sliding window size. Let Module A calls Module B using interface of Module B. The implemented method necessitates interface of module to perform test suite reduction for regression integration testing.

To implement the test suite reduction method, the method names and its signatures that are in the interface module are first extracted .The extracted methods and its signatures are shown in the later section.

The next step is to compare the method names and respective signatures of calling module with the extracted method names and signatures. When matches occur, it indicates that calling module interacts with interfaced module through interface. All such matches are extracted to find the sequences of calls of calling module with interfaced module. Such an extracted sequence is shown in Table 2. Once sequences were extracted and all possible permutations of the obtained sequences are computed to identify correct and incorrect sequence of

Table 1: Method name and signatures of stack module

Method name	Method signatures
Void scapacity	Int size
Void push	Int element
Int pop	No signatures
Int peek	No signatures
Boolean is empty	No signatures

Table 2: Classification of Redundant and Non-Redundant Permutations

Sliding window size	All possible permutations	Redundant permutations	Non-redundant permutations
2	480	460	20
3	360	300	60
4	240	120	120
5	120	-	120

calls. This helps in identifying integration errors. The permutated sequences are shown below. For further process in identifying correct and incorrect sequence of calls easily, all redundant permutated sequences are eliminated and only non-redundant sequences are retained. Those non-redundant sequences are illustrated in Table 3. From the obtained non-redundant sequences, the correct and incorrect sequences of calls are classified and they are used in constructing reduced test suite set.

The above experiment is carried out by fixing the sliding window size as 2,3,4 and 5 for the sequence of length 5 to find the optimum sliding window size. The Table 2 which is shown above depicts the total number of permutations possible for sliding window of size 2,3,4 and 5. It also classifies the total number of redundant and non-redundant permutations possible for each sliding window size.

Table 3 illustrates number of correct and incorrect sequence of calls for each size of the sliding

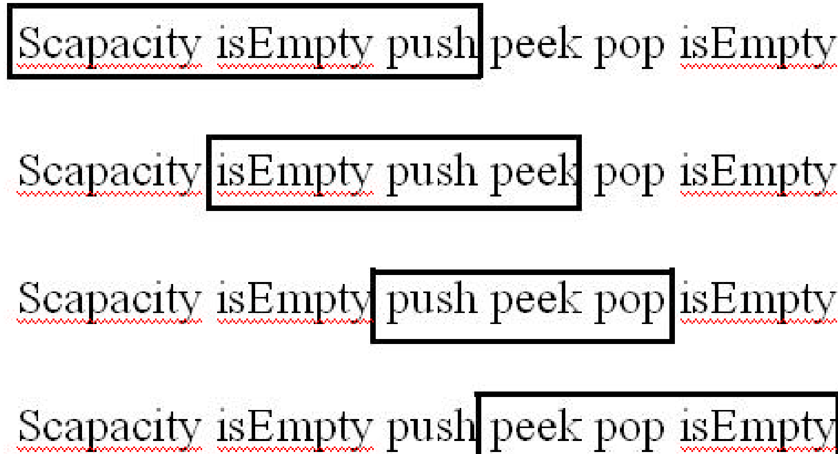

Fig. 2: Levels of sliding window size

Table 3: Classifications of correct and incorrect sequence of calls

Sliding window size	Non-redundant permutations	Sequences of calls	
		Correct sequence	Incorrect sequence
2	20	9	11
3	60	30	30
4	120	30	90
5	120	12	108

Table 4: Siemens test suite

Program name	Lines of Code	No. of versions	No. of functions
Print tokens	726	7	18
Print_tokens2	570	10	19
Schedule	412	9	18
Schedule2	374	10	16
Tcas	173	23	9
Tot_info	565	41	7

Table 5: Functions for “tcas c” program

Function name	Sequences of calls	
	Correct sequence	Incorrect sequence
Alim (1)	2	16
Inhibit Bias climb (2)	1	17
Initialize (3)	2	16
Own_ above _Threat (4)	3	15
Own_below_ Threat (5)	0	18
Non_crossing_bias_ limb (6)	1	17
Non_crossing_Bias _descend (7)	1	17
Main (8)	3	15
Alt_sep_test (9)	3	15

window. Figure 2 shows the correct sequence and incorrect sequence for various window sliding size between 2-18.

From the above result, it is evident that optimum sliding window size suitable to reduce test suite size efficiently for the stack application is 3 and test suite is reduced based on the interaction sequence obtained with optimum sliding window size fixed in the obtained sequences.

For implementing the proposed technique, seven programs from Siemens test suite has been taken. They are print_tokens and print_tokens2 are used as lexical analyzer, replace is used for pattern matching and replace a sting, schedule and schedule2 are for scheduling the jobs, tot_info is for giving statistical information for the given data and tcas is used in aircraft. The Siemens researchers created test pools and faulty versions for each programs. Table 4 shows the details needed for evaluating the technique from Siemens test suite. The resultant sequence function calls for the program tcas.c from Siemens test suite has been shown in Table 5.

CONCLUSION

This proposed method is implemented based on the interaction between sequences of call obtained which reduces the difficulty of source-code dependency strategy for reducing the test suite size. This method is efficiently utilized for all available ready-made software whose source-code is fairly large. This proposed model also overcomes the difficulties of recurrent looping problems which complicates the call stack coverage plan for test suite reduction. This proposed method is more efficient when compared to all previous existing techniques and provides better results.

REFERENCES

Delamaro, M.E., J.C. Maidonado and A.P. Mathur, 2001. Interface mutation: An approach for integration testing. IEEE. Trans. Software Eng., 27: 228-247.

Haley, A. and S. Zweben, 1984. Development and application of a white box approach to integration testing. J. Syst. Software, 4: 309-315.

Harrold, M.J., R. Gupta and M.L. Soffa, 1993. A methodology for controlling the size of a test suite. ACM Trans. Software Eng. Methodol., 2: 270-285.

He, Z.F., B.K. Sheng and C.Q. Ye, 2005. A genetic algorithm for test-suite reduction. Proceesings of the 2005 IEEE International Conference on Systems, Man and Cybernetics, October 12-12, 2005, IEEE, New York, USA., ISBN: 0-7803-9298-1, pp: 133-139.

Hofmeyr, S.A., S. Forrest and A. Somayaji, 1998. Intrusion detection using sequences of system calls. J. Comput. Secur., 6: 151-180.

Kichigin, D.Y., 2007. A Method of Test Suite Reduction. ISP RAS, Moscow, Russia,.

Linnenkugel, U. and M. Mullerburg, 1990. Test data selection criteria for (software) integration testing. Proceedings of the 1st International Conference on Systems Integration, April 23-26, 1990, IEEE, New York, USA., ISBN: 0-8186-9027-5, pp: 709-717.

McMaster, S. and A.M. Memon, 2005. Call stack coverage for test suite reduction. Proc. IEEE. Int. Conf. Software Maintenance, 00: 539-548.

Piwowarski, P., M. Ohba and J. Caruso, 1993. Coverage measurement experience during function test. Proceedings of the 15th International Conference on Software Engineering, May 17-21, 1993, IEEE, New York, USA., ISBN: 0-8186-3700-5, pp: 287-301.

Rothermel, G., M.J. Harrold, J.V. Ronne and C. Hong, 2002. Empirical studies of test suite reduction. Software Test. Verification Reliab., 12: 219-249.

Rountev, A., S. Kagan and J. Sawin, 2005. COVERAGE CRITERIA for Testing of Object Interactions in Sequence Diagrams. In: International Conference on Fundamental Approaches to Software Engineering. Cerioli, M. (Ed.). Springer, Berlin, Germany, pp: 289-304.

Zhu, H., P.A.V. Hall and J.H.R. May, 1997. Software unit test coverage and adequacy. ACM. Computing Survey, 29: 366-427.