A DATA FLOW BASED NOVEL TESTING STRATEGY FOR UNIT TESTING OF OBJECT ORIENTED SOFTWARE

S. Suguna Mallika
Department of CSE,
CVR College of Engineering,
Hyderabad, India.

J. Vamsi V. Krishna
Department of IT,
CVR College of Engineering,
Hyderabad, India.

P.Amulya Sri
Department of CSE,
CVR College of Engineering,
Hyderabad, India.

Abstract: Uncovering errors during unit testing only, lessens the probability of the propagation of errors to other phases in a large number. While this fact is applied to object oriented software, it is understood that the fundamental units with object oriented software are precisely the classes and hence the classes need to be thoroughly tested to accomplish unit testing. Testing of a class is analogous to testing the methods defined as part of the class. While it is known that the various methodologies to testing conventional software are path testing, transaction flow testing, data flow testing et. al, an attempt has been made in the current work to use the data flow testing technique partially to come up with a novel proposal so as to help the independent unit tester decide on the most important methods for testing within the class. The strategy would assist the tester in deciding on the priority of methods to be tested and thereby save on the testing effort.

Keywords: data flow testing, unit testing, OO testing, testing strategy, du pairs

1. INTRODUCTION

In the industry it is not an uncommon sight that testing has to be done frequently on every new release of the software. While we conduct regression tests in order to ensure that the existing functionality does not get affected there is a continuous pursuit to optimize the testing effort and time. One way to optimize the testing time is to execute lesser number of test cases while simultaneously achieving the same correctness of software as with running all the test cases. For unit testing Object Oriented Software an attempt has been made to come up with a strategy to test the individual classes. Performing Unit testing on classes is analogous to performing the individual methods testing which have been defined in the class.

Running every test case for all the methods would be quite time consuming when there is an urgency to determine the health of the class. In such contexts if the most important methods out of all the methods could be prioritized then those high priority methods could be subjected to full logic coverage testing while the rest of them could be subjected to some black box testing techniques like Equivalence Partitioning, Boundary Value Analysis etc.

2. PROPOSED STRATEGY

The proposal is that all possible du-chaining of the data members defined in the class is done. Then a method which is the most repeated in the majority of the du chains is assigned a high priority factor and is subjected to logic coverage testing. The remaining sequences of methods are tested under normal conditions using state based testing techniques applying on the constraints of the object's state. For arriving at assigning priority of the methods, i.e to understand the most important methods of all, an algorithm has been proposed. The algorithm comprises of the following steps:

Step 1: def-use pairs of all the data members defined in the class are made.

Note: A def-use pair corresponds to the listing of a pair of line numbers where the first number indicates the line of occurrence of the definition of the variable and the second number indicates the line number where usage of the variable is seen without the variable getting killed in between.

Step 2: The pairs of line numbers are mapped to method names in which the line numbers are occurring thus arriving with the set of method names in which the definition of a data member is occurring and the method name in which it is being used subsequently.

Step 3: Now an individual counter is maintained against each method and du pairs of methods is observed.

Step 4: With the occurrence of each method in the listing, the counter against that corresponding method is incremented.

Step 5: Step 4 is repeated until all the methods listed in the du pairs are exhausted.

Step 6: Now for each of the methods a table is prepared with the final counter value against it.

Step 7: The table is sorted based on the counter value for each method.

Step 8: Starting with the maximum counter value priority is assigned in the increasing order. I.e method with the maximum counter value is assigned a priority of 1 and so on.

Step 9: If more than one method arrive at the same counter value same priority is assigned to both the methods.

Step 10: After assigning the priority the method with the highest priority is subjected to full logic coverage testing while the other methods are subjected to equivalence partitioning and boundary value analysis testing.

3. CASE STUDY

The above algorithm has been manually traced on a sample case study i.e a stack array class under consideration. Code for the Stack Array Class which has methods like push, pop, top, peek, isEmpty, isFull, getSize. The sample code for the class written is as follows:

```
1.public class StackArray {
2.
3.      private int[] stackElements;
4.      private int topOfStack;
5.      private int capacity;
6.      private int size;
7.
8.
9.      public StackArray(){
10.             this(30);
11.     }
12.
13.     @SuppressWarnings("unchecked")
14.     public StackArray (int capacity) {
15.             this.capacity = capacity;
16.             size                    = 0;
17.             topOfStack              = -1;
18.             stackElements = (int[]) new
        Object[capacity];
19.     }
20.
21.     @Override
22.     public boolean isEmpty() {
23.
24.             return size == 0;
25.     }
26.
27.     @Override
28.     public boolean isFull() {
29.
30.             return size == capacity;
31.     }
32.
33.     @Override
34.     public void push(int dataIn) throws
35.StackOverflowException {
36.
37.             if(isFull())
38.     throw new StackOverflowException();
40.     topOfStack = topOfStack +1;
41.     stackElements[topOfStack] = dataIn;
42.             ++size;
43.
44.     }
45.
46.     @Override
47.     public int pop() throws
        StackUnderflowException {
48.
49.             if(isEmpty())
50.throw new StackUnderflowException();
51.
52.int dataOut =stackElements[topOfStack];
53.     topOfStack = topOfStack -1;
54.     --size;
55.     return dataOut;
56.     }
57.     @Override
58.public int peek() throws
StackUnderflowException{
59.
60.if(isEmpty())
61.throw new StackUnderflowException();
62.
63.     return stackElements[topOfStack];
64.     }
65.
66.     @Override
67.     public int getSize() {
68.
69.             return size;
70.      }
71. }
```

Computation of the DU Pairs for each data member defined in the class:

The first number specifies the line of occurrence of the definition of the data member and the second line signifies the usage of the data member. On the right hand side is the listing of the method in which the definition has first appeared followed by the method name in which the subsequent usage has occurred.

For data member StackElements :

(41,52) – (push(),pop())
(41,63) – (push(), peek())

For data member topOfStack :
(17,40) – (StackArray(), push())
(40,41) - (push(), push())
(40,52) – (push(), pop())
(40,53) – (push(), pop())
(53,63)- (pop(), peek())

For data member size :
(16,24) – (StackArray(),isEmpty())
(16,30) – (StackArray(),isFull())
(16,42) – (StackArray(),push())
(42,54) – (push(), pop())
(54,69) – (pop(), getSize())

For data member capacity :
(15,18) – (StackArray(),StackArray())
(15,30) – (StackArray(),isFull())

4. RESULT ANALYSIS

Table 1: Priority Computation Table:

S.No	Method Name	Count	Priority
1	StackArray	7	2
2	push	9	1
3	pop	6	3
4	isFull	2	4
5	isEmpty	1	5
6	getSize	1	5
7	peek	2	4

After the priority computation for each of the methods, as per the strategy the method push () has to undergo full logic coverage testing. And the remaining methods would be subjected to Equivalence Partitioning or Boundary value Analysis testing techniques.

5. RELATED WORK

Testing Object Oriented Software is an area where currently lot of research is going on. However it is a common observation that many of the techniques resort to high complexity algorithms [3, 4, 6, 7 and 9]. They are often difficult to implement and execute when in times of emergency to check the health of the class. Our technique is based out on employing a simpler algorithmic approach to find out the most important focus area in the class and thus help the tester with subtle inputs to quickly assess the health of the class.

6. CONCLUSIONS

By following this strategy, the following drawbacks are overcome which are present with the existing strategies:

- Computational complexity of techniques like symbolic execution and automated deduction.
- Laborious time involved with robust testing techniques.
- The drawback of model-driven testing is that the test is only as good as the model and we know from practical experience that models are seldom complete and most often inconsistent. This again is an approach which is theoretically appealing but does not hold up in practice.
- This strategy helps in saving testing time while delivering quality software.

7. ACKNOWLEDGEMENTS

Our sincere thanks to Mr.B.Vikranth, Associate Professor, Department of IT, CVR College of Engineering for his inundated support all through in the development of this paper.

8. REFERENCES

[1] Roger S. Pressman, Software Engineering A Practitioner's Approach, Seventh Edition, McGraw-Hill Int'l Edition.

[2] Boris Beizer, Software Testing Techniques, Second Edition, International Thomson Computer Press, 1990.

[3] Harry M. Sneed, "Testing Object Oriented Software Systems", Proceeding ETOOS '10 Proceedings of the 1st Workshop on Testing Object-Oriented Systems ACM New York, NY, USA

[4] Ugo Buy, Alessandro Orso and Mauro Pezze, "Automated Testing of Classes" ISSTA '00, ACM, Portland, Oregon.

[5] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer, "ARTOO: Adaptive Random Testing for ObjectOriented Software", ICSE'08, May 10–18, 2008, Leipzig, Germany, ACM 9781605580791/08/05.

[6] Lucas and Serpa Silva, "Evolutionary Testing of Object-Oriented Software" ACM SAC'10, March 22-26, 2010, Sierre, Switzerland, 978-1-60558-638-0/10/03.

[7] HUO YAN CHEN, T. H. TSE and T. Y. CHEN,"TACCLE: A Methodology for Object-Oriented Software Testing at the Class and Cluster Levels" ACM Transactions on Software Engineering and Methodology, Vol. 10, No. 4, January 2001, Pages 56–109.

[8] S.R.Chidamber and C.F.Kemerer, "A Metrics Suite for Object Oriented Design", IEEE Transactions on Software Engineering, Vol. 20, No. 6, 1994, pp. 476-493.

[9] Mauro Pezze and Michal Young, "Testing Object Oriented Software", IEEE Proceedings of the 26th international conference on Software Engineering(ICSE' 04).

[10] http://www.guru99.com/software-testing-life-ycle.html.