Search-Based Test Data Generation for Object- Oriented Programming

R.P.Mahapatra[1] A.Kulothungan[2] Priyanka Sachdeva[3] Pratibha Dabas[4]
[1]Professor & HOD [2]Assistant [3,4]Professor
[1,2,3,4]Department of Computer Science & Engineering
[1,2,3,4]SRM University

Abstract— Test-data generation for object-oriented programming (OOP) is challenging due to the features of OOP, e.g., abstraction, encapsulation, and visibility that prevent direct access to some parts of the source code. To address this problem we present a new automated search-based software test-data generation approach that achieves high code coverage for the source code and reduces the search space. For that we first describe the test-data generation problem for object oriented testing to generate relevant sequences of method calls. In this paper, proposed technique uses means-of-instantiation, diversification strategy and seeding strategy for object oriented generation and sequence caller for search space reduction. Finally, we show that our technique is more efficient than the randomized technique as it reduces the search space better than the randomized one. It also uses a seeding strategy and a diversification strategy to increase the likelihood to reach a test target.

Key words: Oriented Programming, Based Test Data Generation

I. INTRODUCTION

A. Software Testing

Software testing involves the execution of a software component or system component to evaluate one or more properties of interest. It's the most important part of the software development also time consuming and tedious process. It's most expensive part of testing is test data generation. Due to the complex features of object-oriented programming languages (OOP), e.g., abstraction, encapsulation, and visibility it prevents the direct access to some parts of the source code. To address this problem, a test-data generator is needed to perform(1) instantiation of the classes; (2) perform a sequence of method calls to put the instance of the class under test in a desired state (i.e., a state that may help to reach the test target); and, (3) call a method that may reach the test target.

B. Search-Based Software Testing

We are using the Search Based Software Testing (SBST) to address the problem of automating test-data generation. Search-Based Software Testing is the use of a meta-heuristic optimizing search technique, such as a Genetic Algorithm, to automate or partially automate a testing task; for example the automatic generation of test data. SBST has been successfully applied to solve the problem of test-data generation which translates it into a search problem by providing a feasible solution to the original problem and searching an actual solution using a search heuristic.

C. Seeding Strategy

For each primitive data type or string, it collects constants from the source code, generates new values, then seeds them while generating data. It defines a seeding probability for each data type and each constant according to the number of collected occurrences of the constants. Also, it seeds the null constant with a constant probability while generating instances of classes.

D. Diversification Strategy

This strategy generates the needed instances of a given class by using different means of- instantiations. The number of reuses of a same means-of-instantiation depends on its complexity and computes a representative complexity measure for each means of- instantiation. Initially, it supposes that any class requires a constant complexity to be instantiated and a means-of-instantiation requires the total complexity of its arguments, then it dynamically adjusts this measure at each attempt of instantiation.

II. PROBLEM DESCRIPTION

A. Existing Problem

The earlier techniques used to reduce the search space was the random technique. With random approach, the search space is large because of four reasons:

1) There is no restriction on methods to call;
2) There is no restriction on the length of sequences of method calls;
3) The order of method calls is undefined;
4) The possible instances of a class under test or an argument may be "unlimited".

B. Proposed Solution

To address above problem we present a new automated search-based software test-data generation approach that achieves high code coverage for the object oriented code and reduces the search space.

So, in this paper we are doing the following-

1) Identify the test data generation problem in unit testing.
2) Generate the instances of the classes.
3) Test data generation of unit testing.
4) Block Diagram

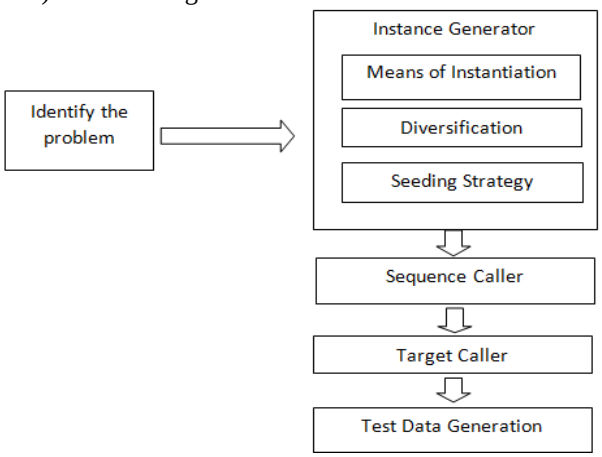

Fig. 1: Overall Diagram

Fig. 1 shows the overall diagram of our proposed approach. Firstly, we identify the problem i.e, the source code of the object oriented program. Then through the help of instance generator we create the means of instantiation, using diversification strategy and seeding strategy for removing the existing problem from our source code. Then by this output we call the sequence caller to generate the sequence of the methods. The output of sequence caller now becomes the input of the target caller to call the method having our target. Now output of target caller is used as the input for the test data generation to generate the required test data for the object oriented classes.

III. METHODOLOGY

Our proposed technique uses different methodologies:-

1) Instance generator
2) Sequence caller
3) Test data generation

A. Instance Generator

A test-data generation problem is an instantiation of the Object oriented class and a sequence of method calls on that instance. Calling a constructor or a method requires some instances of classes. The instance generator implements the generation of means-of-instantiation, the diversification strategy and theseeding strategy.

Algorithm 1. Instance Generator

Input: Set of instances for every classes
Output: Set of instances for dependent classes

1. C_i ={$C_1, C_2, \ldots\ldots C_n$} // set of classes
2. t = target_task
3. // generate instance for dependency class for target dc_i = {$dc_1, dc_2, \ldots\ldots dc_n$}
4. **for** (i= 1 to n)
5. O_i = get instance()
6. **for** every dc_i
7. $instance_0 = 0$
8. **If** (dc_i dependent on target)**then**
9. $instance_i$=$instance_{i-1}$ ∪ {$instance_i$}
10. **else**
11. reject dc_i
12. **return**$instance_i$

Fig. 2: Instance Generator

B. Sequence Method Generator

The sequence method caller is used to call the sequence from the available sequences, according to which the methods will be called, unlike the random approach which called any method without any sequence. The following is its algorithm.

Algorithm 2. Sequence method call

Input: Set of dependent classes
Output: Sequence of dependent methods

1. // Generate all possible sequences
2. n= total dependent class
3. $T_0 = \Phi$
4. **for** (i= 1 to n)
5. $T_i = T_{i-1} \cup$ (set of task in C_i)
6. // Generate random sequence of methods
7. ST= rand(T_i) , where i= 1 to n
8. // Finding dependent sequence
9. DC =0
10. DM=0
11. // Start in bottom up manner
12. **while** (target_task in class != base class)
13. find(v) = variable in the class
14. **for**(every variable in find(v)) {
15. class =class of variable
16. DC= DC ∪ {new DC}
17. DM = DM∪ {new DM of variable} }
18. end **while**
19. **return** DM, DC

Fig. 3: Sequence Method Call

C. Test Data Generator

This component operates and coordinates other components to generate test data by calling the method having the target method.It implements the skeleton of the whole process of test data generation.
The following is the algorithm for test data generation.

Algorithm 3. Test data generation

Input: Set of values
Output: Test cases and reduced search space

1. // Generate overall test case
2. TSU = Φ
3. **for** (every $class_i$ in dependent class set) {
4. **for** (each object (i= 1 to n))
5. $TSU_{i,j} = TSU_{i,j} \cup$ {each value of ob_i} }
6. **return** $\sum TSU_{i,j}$,
   where; i= no. of classes from 1 to n
   j= no. object in each class from 1 to n
7. // random method sequence
8. **for** (every $class_i$ in randomly choosen by dependent class) {
9. **for** (each object (i= 1 to n))
10. $TSU_{i,j} = TSU_{i,j} \cup$ {each value of ob_i} }
11. return $\sum TSU_{i,j}$,
12. // dependent sequence
13. **for** (every $class_i$ of dependent class) {
14. **for** (each object (i= 1 to n))
15. $TSU_{i,j} = TSU_{i,j} \cup$ {each value of ob_i} }
16. **return** $\sum TSU_{i,j}$

Fig. 4: Test Data Generation

IV. CASE STUDY

The following is the case study that we are considering in our research paper.

A. Hybrid Inheritance

Our case study has the hybrid inheritance in which there are six classes, having base classes and derived classes which

are inherited from the super class. Each class hasit's own functions, tasks and variables and also inherited from above base classes. A is the superclass B and C classes are derived from it, D is derived from both B and C so has functions and variables of both the classes, E and F classes are derived from class D. Class A has the task 1, B has task 2, class C has task 3, class D has task 4,5,6, E class has task 8,9 and class F has task 10,11,12. Each tasks having functions and variables which are either independent or are dependent on other base classes.

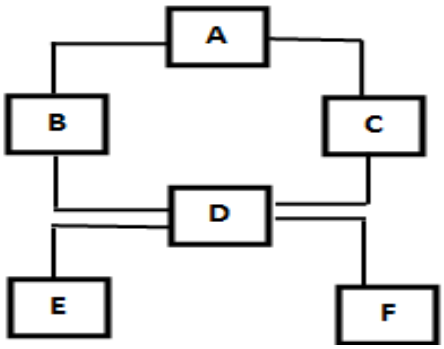

Fig. 6:

B. Overall Test Generation

There are six classes each having two instances and every instance has three values and we can generate all possible test cases for them. The test cases that we have evaluated are 658 upto class D and 3564 test cases upto class E and after removing the common ones we get 2916 test cases. The total test cases are 6480.

C. Instance Generator

Here the instance generator algorithm is used where the input is the set of instances for every class. Now generateinstances of dependent classes for target and the output will contain the set of instances for the dependent classese.g, class A has two instances, each instance having three values and functions needed to get target. We have taken the target in class E. The total test cases needed to achieve the target is 3564.

D. Sequence Generator

The constructor level and functional level dependencies are considered here.

In the **constructor level dependency**, we have taken the constructors of the classes and the dependencies are checked with the other classes. Class A has constructor A() having task 1, which is then used by classes B and C having their constructors B() and C() with task 2 and 3 respectively. Like this $F(T_{10})$, $E(T_7)$, $D(T_4)$ and $D(T_5)$ dependent on $B(T_2)$ which is dependent on $A(T_1)$. $D(T_2)$ and $F(T_{11})$ dependent on $C(T_3)$ which is dependent on $A(T_1)$. $E(T_9)$ dependent on $D(T_5)$. $E(T_8)$, $F(T_{11})$, $F(T_{12})$ dependent on $D(T_6)$, only $A(T_1)$ is independent as it's the super class.

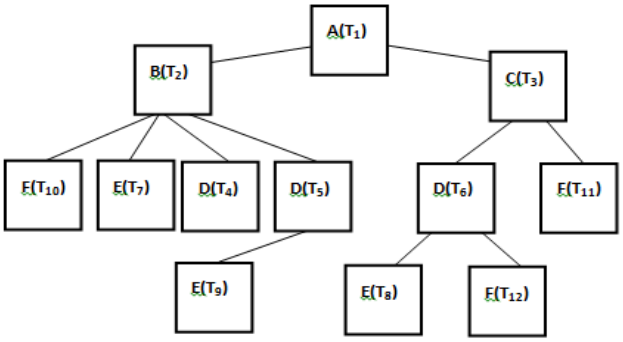

Fig. 7: Constructor level dependency

Random dependency structure is taken for the target task which randomly selects the sequence of methods which leads to the target task. In our case we have two sequence of methods leading to the target task (Fig.4), the target is present in the class F, each class has two objects, two independent paths in the random dependency from which random dependency will choose any one of these two paths.

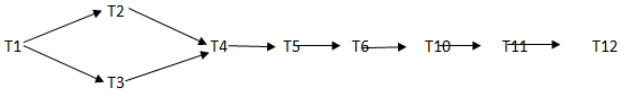

Fig. 8: Random Dependency

Two possible sequences are as follows

T1 → T2 → T4 → T5 → T6 → T10 → T11 → T12

Fig. 9: Path 1

T1 → T3 → T4 → T5 → T6 → T10 → T11 → T12

Fig. 10: Path 2

Dependency Sequence callerwhich takes only the functions which are dependent on other class functions. In our case there is only one such path which is as follows. The targetis in the T11 ,so we perform sequence caller in bottom up manner to check whether the target class has variables of the above base classes, if it has then again check that class for the variables of above class and like this proceed.

The algorithm that we use here is the sequence method call which has the set of dependent class as the input. Generating all the possible sequences by the random sequence method and selecting any sequence from the possible sequences and by dependent sequence method and checking the target in the class and variables of the above classes. The output will be the sequence of dependent methods.

T1 → T3 → T11

Fig. 11: Path 3

E. Test Data Generation

Test data generation using random method and dependency method.

Random Method for path 1			Random Method for path 2			Dependency Method		
Classes and Tasks	Total Test Cases	Classes and Tasks	Total Test Cases	Search space for report	Search space for report	Classes and Tasks	Total Test Cases	Search space for report

$A(T_1)$	12	$A(T_1)$	12	12	12	$A(T_1)$	12	12
$B(T_2)$	36	$C(T_3)$	36	36	36	$C(T_3)$	36	36
$D(T_4, T_5, T_6)$	126	$D(T_4, T_5, T_6)$	126	378	378	$D(T_4, T_5, T_6)$	126	378
$E(T_7, T_8, T_9)$	1296	$E(T_7, T_8, T_9)$	1296	3888	3888	$E(T_7, T_8, T_9)$	1296	3888
∑(test cases & search space)	1470	∑(test cases & search space)	1470	4314	4314	∑(test cases & search space)	1470	4314

Table 1: Random Generation and Dependency Generation

V. GRAPH ANALYSIS

The table 2 shows the number of tests shoots, test cases and search space.

Test shoots	Test Cases	Search Space
Overall	3564	6069
Random 1	1470	4314
Random 2	1470	4314
Dependency	300	300

Table 2: Test cases

Fig. 9 and Fig. 10 are the two graphs showing the test shoots, where,
TS1 – overall test shoot,
TS2 – random test shoot for path 1,
TS3 - random test shoot for path 2 and
TS4 – dependency test shoot

Fig. 9 graph shows number of test cases on y axis and test shoots on x axis. TS1, TS2, TS3 and TS4 has 3654, 1470, 1470 and 300 test cases respectively. Fig. 10 graph shows number of search space on y axis and test shoots on x axis.TS1, TS2, TS3 and TS4 covers 6069, 4314, 4314 and 300 search spaces respectively.

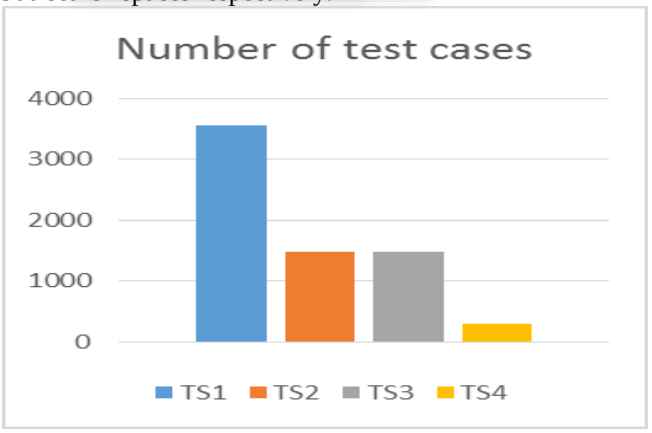

Fig. 12:

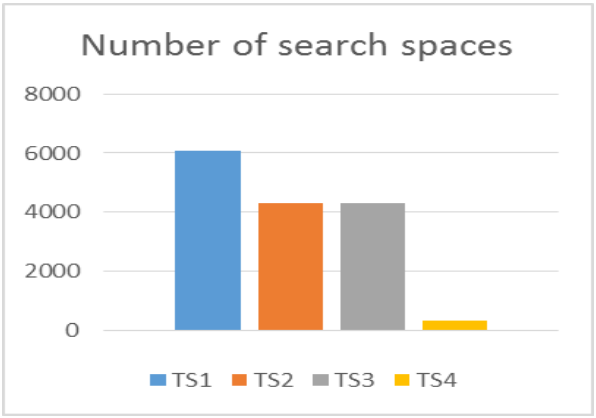

Fig. 13:

VI. CONCLUSION AND FUTURE WORK

Earlier search-based software testing has been used to solve the problem of automated test data generation for procedural programming as well as for object-oriented programming. Yet, test-data generation for OOP is challenging due to the features of OOP, e.g., abstraction, encapsulation, and visibility that prevent direct accessto some parts of the source code. In this paper we have presented a new automated search-based software testdata generation approach that achieves high code coverage for the source code and reduces the search space.

In our research paper we have considered the function dependency and class dependency but not considered the access modifier and abstractions, so in future the research can be focused on the above object oriented feature for reducing the search space.

REFERENCES

[1] AbdelilahSakti, Gilles Pesant, and Yann-Ga€elGu_eh_eneuc, Senior Member, IEEE "Instance Generator and Problem Representation to Improve Object Oriented Code Coverage" in IEEE Transactions on software engineering IEEE, vol. 41, no.3, March 2015
[2] N. Alshahwan and M. Harman, "Automated web application testing using search based software engineering," in Proc. 26th IEEE/ACM Int. Conf. Autom. Softw. Eng.,2011,pp. 3–12.
[3] M. Alshraideh and L. Bottaci, "Search-based software test data generation for string data using program-specific search operators,"Softw. Testing, Verification Rel., vol. 16, no. 3, pp. 175–203,2006.
[4] J. H. Andrews, S. Haldar, Y. Lei, and F. C. H. Li, "Tool support for randomized unit testing," in Proc. 1st Int. Workshop Random Testing, 2006, pp. 36–45.
[5] J. H. Andrews, T. Menzies, and F. C. Li, "Genetic algorithms for randomized unit testing," IEEE Trans. Softw. Eng., vol. 37, no. 1,pp. 80–94, Jan./Feb. 2011.
[6] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," Softw. Testing, Verification Rel., vol. 24, no. 3, pp. 219–250, 2014.
[7] A. Arcuri and G. Fraser, "On parameter tuning in search basedsoftware engineering," in Proc. 3rd Int. Conf. Search Based Softw.Eng., 2011, vol. 6956, pp. 33–47.

[8] A. Arcuri and X. Yao, "Search based software testing of object oriented containers," Inf. Sci., vol. 178, no. 15, pp. 3075–3095, 2008.
[9] S. Barbey and A. Strohmeier, "The problematics of testing object oriented software," in Proc. 2nd Conf. Softw. Quality Manage., 1994, vol. 2, pp. 411–426.
[10] D. Binkley and M. Harman, "Analysis and visualization of predicate dependence on formal parameters and global variables," IEEE Trans. Softw. Eng., vol. 30, no. 11, pp. 715–735, Nov. 2004.
[11] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "Artoo," in Proc. 30th Int. Conf. Softw. Eng., 2008, pp. 71–80.
[12] L. Clarke, "A system to generate test data and symbolically execute programs," IEEE Trans. Softw. Eng., vol. SE-2, no. 3, pp. 215– 222, Sep. 1976.
[13] C. Csallner and Y. Smaragdakis, "Jcrasher: An automatic robustness tester for Java,"Softw.:Pract. Exp., vol. 34, no. 11, pp. 1025– 1050, 2004.
[14] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," in Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng., 2011, pp. 416–419.
[15] G. Fraser and A. Arcuri, "The seed is strong: Seeding strategies in search-based software testing," in Proc. IEEE 5th Int. Conf. Softw. Testing, Verification Validation, 2012, pp. 121-130.
[16] G. Fraser and A. Arcuri, "Evosuite at the SBST 2013 tool competition," in Proc. IEEE Int. Conf. Softw. Testing Verification ValidationWorkshop, 2013, pp. 406–409.
[17] G. Fraser and A. Arcuri, "Whole test suite generation," IEEE Trans. Softw. Eng., vol. 39, no. 2, pp. 276–291, Feb. 2013.
[18] G. Fraser and A. Zeller, "Exploiting common object usage in test case generation," in Proc. IEEE 4th Int. Conf. Softw. Testing, Verification Validation, 2011, pp. 80–89.
[19] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," IEEE Trans. Softw. Eng., vol. 38, no. 2, pp. 278–292, Mar./Apr. 2012.
[20] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," ACM SIGPLAN Notices, vol. 40, pp. 213–223, Jun. 2005.
[21] M. Harman, C. Fox, R. Hierons, L. Hu, S. Danicic, and J. Wegener, "Vada: A transformation-based system for variable dependence analysis," in Proc. 2nd IEEE Int. Workshop Source Code Anal. Manipulation, 2002, pp. 55–64.
[22] M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, and J. Wegener, "The impact of input domain reduction on search-based test data generation," in Proc. 6th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng., 2007, pp.155–164.
[23] B. Korel, "Automated software test data generation," IEEE Trans. Softw. Eng., vol. 16, no. 8, pp. 870–879, Aug. 1990.
[24] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," Ann.Math. Stat., vol. 18, pp. 50–60, 1947.
[25] P. McMinn, "Search-based software test data generation: A survey," Softw. Testing Verification Rel., vol. 14, pp. 105–156,2004.

