2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)

Automatic Test Case Generation and Optimization Based on Mutation Testing

Yunqi Du
Dept of Computer Science and Technology
Southwest University of Science and Technology
Sichuan, China
631135138@qq.com

Haiyang Ao
Dept of Computer Science and Technology
Southwest University of Science and Technology
Sichuan, China
oceanao@163.com

Yong Fan
Dept of Computer Science and Technology
Southwest University of Science and Technology
Sichuan, China
fany@swust.edu.cn

Ya Pan
Dept of Computer Science and Technology
Southwest University of Science and Technology
Sichuan, China
panya@swust.edu.cn

O.ALEX
Dept of Computer Science and Technology
Southwest University of Science and Technology
Sichuan, China
3261916694@qq.com

***Abstract*—Based on defect implantation mutation testing technique not only serves as a standard for evaluating test cases but also guides how to generate high-quality test case sets. In order to reduce the number of mutants, we propose a mutation operator selection strategy according to Selective Mutation. From 19 mutation operators of Mujava we select 5 mutation operators to obtain a subset. Test cases using this subset are able to achieve an average variation score of more than 95% on the variants of the complete set. Then we propose a test case generation method combining mutation testing with a genetic algorithm. The crossover, insertion, change, and deletion operators of the test case set are redefined, and the test cases are optimized. Finally compared with some algorithms and tools we obtain a set of test cases with higher coverage and higher mutation score.**

Keywords- test case generation, mutation testing, mutation operator, genetic algorithm.

I. INTRODUCTION

Software testing is a basic means of assessing the quality of software products. How prove theoretically or practically whether one has done enough testing is still a classic challenge in software testing. To resovle this challenge, researchers usually set a coverage criteria to evaluate the test cases. These coverage criterias measure the extent to which testers conduct tests in a certain direction. However, some researchers have theoretically proved in the scientific sense based on Popper that these forms of coverage are unreliable [1].

Mutation testing provides a mechanism for these challenges that can be used to prove the problem of test efficiency. According to the calculation formula, the mutation score is calculated to evaluate the test case, which can be used to measure the bug detection ability of the test case throughout the test activity, and assess the adequacy of test cases in the current testing. The full coverage of the variability test is still undecidable[2].

The execution number of test cases in the mutation test is a very large number, which is the product of the number of mutants and use cases. For Java programs, based on the weak mutation testing transformation method, we extract the appropriate information from the Java bytecode to complete the construction of the original program control flow chart, and then convert the test case generation problem of the coverage branch into the problem of minimizing the function Fitness problem. We design asserted auto-generated strategies and design test case optimization methods to optimize the resulting test cases. We have experimentally verified the proposed mutation testing set generation method in this paper, and compared it with other existing algorithms and existing popular tools.

II. MUTATION OPERATORS SELECTION

Offutt et al. classified 22 mutation operators into three categories according to the differences of grammatical function [3], which included operator substitution class, expression modification class and statement modification class. They proved mutation operators ABS、AOR、LCR、ROR、UOI have ProbSubsumes relationship for 22 mutation operators. It showed that the mutation operators of the expression modification class has the ability to represent the whole set of mutation operators in their experiments, and got relative mutation score was 99.51 for all mutation operators.

According to the above experimental method, mutation operators in MuJava are classified and six of the expression modification classes AOIU, AOIS, ROR, AORS, AORB, LOR are selected as subclasses of Java mutation operators.

978-1-7281-3925-8/19/$31.00 ©2019 IEEE
DOI 10.1109/QRS-C.2019.00105

We propose an algorithm as Fig.1 and obtain the result that five mutation operators AOIU、AOIS、ROR、AORS and AORB can satisfy demand of mutation testing for general Java program.

```
Function 19Mscore              //Obtain the variation score for the 19M combination
    T←0
    If(em exist||score（T，6M）<100）then   //Stop expansion at 100 points
        Remove em                  //Manually remove the equivalent mutant
        T++                        //Extend test case set T
    end if
    return score(T,19M)
end function
Function verification              //Verify the adequacy of 6M set
    for m in 6M
        T1←0
        Remove m                       // 5M
        If(em exist||score（T1，5M）<100）then
            Remove em                  //Manually remove the equivalent mutant
            T1++                       //Extend test case set T1
        end if
    Return score(T1，6M)
end function
```

Figure 1. Mutation Operator Selection

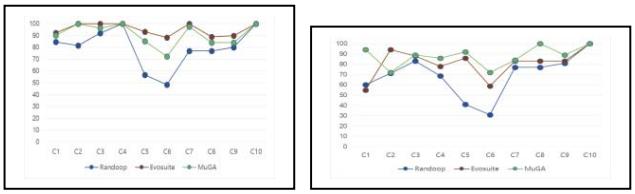
III. TEST CASE GENERATION

We use the algorithm shown in Figure 2 to perform the iterative process of the population. The first line represents the random algorithm to generate an initial population set to M size. The second row calculates and ranks the fitness of

```
1   random generate p(m)      //Initial population
2   p1，p2=p(m).select        //Choose two parents based on the ranking
3   if  random  >  Pc    // crossover operation
4       c1,c2=p1,p2.crossover
5   else
6       c1,c2=p1,p2
7   if  random  >  Pm    // mutation operation
8       c1,c2=p1,p2.mutate
9   else
10      c1,c2=p1,p2
    /*Calculate the parent fitness and length*/
11  Fp=min(fit(p1),fit(p2))    Lp=length(p1)+length(p2)
    /* Calculate the fitness and length of the children*/
12  Fc=min(fit(c1),fit(c2))    Lc=length(c1)+length(c2)
13  if  Fc≤Fp ^^ (Fc=Fp&&Lc≤Lp)
14      p(m)=p(m)∪{c1,c2}    // Return the union of the optimal children
15  else
16      p(m)=p(m)∪{p1 or p2}    // Return the union of the optimal parent generation
```

Figure 2. Test Case Generation Algorithm

each individual. From line 3 to line 10 the genetic algorithm performs routine operations. The mutation strategy of lines 6-10 is a random position mutation strategy. Lines 11-16 represent representative operations within the population. When a target can be covered, multiple test cases may be generated. The population update strategy is to compare and evaluate the test cases. The evaluation rule is the minimum fitness fit of the test case. When the fitness of the two generations is equal, the test case length (the number of call statements) is calculated, and the lower length is optimal. Record the individual that are evaluated as optimal at the time and store them in the set, iterating the next population. When the stop condition C is reached, the loop is exited and the iteration is completed.



(a)Coverage Comparison (b)Mutation Score Comparison

Figure 3. Experiment Result

IV. EXPERIMENT RESULT

In order to obtain reasonable empirical results, this article uses the Java project in SF100, numbered in the order of C1-C10. Our experiment is compared with popular tools Randoop and Evosuite. We use MUGA to represent the algorithm of this paper.

From Figure 3(a), it can be seen that Evosuite outperforms Randoop and MUGA in coverage, and Evosuite's average test case length is 12.8, MUGA's average test case length is 7.2. It can be seen that the test case optimization algorithm in this paper has certain effectiveness in reducing the length of test cases.

Figure 3(b) shows the mutation scores of the three tools. MUGA performs well. Because the random method Randoop does not consider the mutation test at all, the generated test cases are most difficult to kill the mutant. The core algorithm of Evosuite is MOSA, which is a multi-objective algorithm.

V. CONCLUSION

To reduce the number of mutants, we propose a mutation operator selection strategy and obtain a subset of mutation operators, which is able to achieve an average variation score of more than 95% on the variants of the complete set.We propose a test case generation and optimized method combining mutation testing with a genetic algorithm. Compared with some algorithms and tools we obtain a set of test cases with higher coverage and higher mutation score.In the near future we continue to research more effective test cases generation combining mock technology and test assert technology.

REFERENCES

[1] Papadakis M. mutation testing Advances: An Analysis and Survey. Advances in Computers, 2018.

[2] M. Papadakis, Y. Jia, M. Harman, Y. L. Traon, Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique, in 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, pp. 936–946.

[3] Offutt A J, Lee A, Rothermel G, et al. An experimental determination of sufficient mutant operators. Acm Transactions on Software Engineering & Methodology, 1996, 5(2):99-118.

[4] Kintis M, Papadakis M, Papadopoulos A. How effective are mutation testing tools? An empirical analysis of Java mutation testing tools with manual analysis and real faults. Empirical Software Engineering, 2017(8):1-38.

[5] Ma YS, Offutt J, Yong R K. MuJava: an automated class mutation system[J]. Software Testing Verification & Reliability, 2010, 15(2):97-133.