IJESRT
INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY

NEXT GENERATION SMART MUTATION TESTING FOR JAVA

Prof. V. M. Lomte, Mitesh Rampariya, Barkha Sethia, Vishakha Navandar, Lubdha Dahale
Department of Computer Engineering, RMD Sinhgad School of Engg, Pune, India

ABSTRACT

Software testing is a very time consuming process of software development life cycle. The software tester has to think a lot before he generates any test cases. Even after generating the test cases there is no proof that those test cases can actually uncover all the bugs and there is no guarantee of code coverage. The cost of bug also increases drastically as the software is being developed. Hence this paper tries to reduce the stress of testers as well as reduce the cost of bugs by early detection of bugs by implementing mutation testing strategy with new mutation operators introduces in this paper. This will evaluate the quality of test cases and the tester can modify his test cases based on the mutation score generated in order to improve his test cases.

KEYWORDS: Mutation testing, Cost of bugs, Quality of test cases, Mutation score, early detection.

INTRODUCTION

Mutation testing is fault based testing strategy where the bugs are injected in the source code and the test cases are run to find whether the test cases are able to detect those bugs or not. The bugs are injected by the use of mutation operators and the quality of mutation testing completely depends on what operators we use to create mutants of the program to be tested. This paper introduces the technique used in mutation testing for java language with some traditional operators as well as new operators that are not implemented in the existing systems. The secondary focus is on multithreaded programs written in java.

MATERIALS AND METHODS

For developing this software we have used junit package of java and jdk version 1.8. This software takes a sample java program as input and test cases generated by the tester as input, it than evaluates the correctness of the program and the quality of the test cases as well and generates appropriate results based on the mutation score generated for each test case. Finally the tester makes required changes in his test cases in order to improve the quality of his test cases. The mutation testing starts with the first code itself as soon as it is constructed by the developer, this helps to identify all the possible bugs in the early stage of software development life cycle that may occur in the future where the cost of bugs increases rapidly. One of the non-functional requirements of this paper is to increase the knowledge and experience of the tester in generating right test suite which will decrease the time consumption in testing the software

Mutation Operator Implementation

The mutation operators can be implemented by analysing the input program and the available mutation operators to calculate the implementable locations in the source code. These locations are some instructions that contains the operators and operands or only operators which can be modified by the mutation operators present in the library. An abstract syntax tree can be used to identify the location for the mutation operator in the source code as follow.

Implementation of operator AOR (Arithmetic Operator Replacement) in existing system.

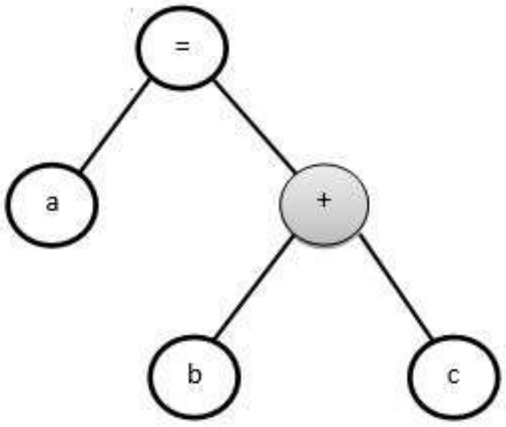

Figure 2.1 Original instruction

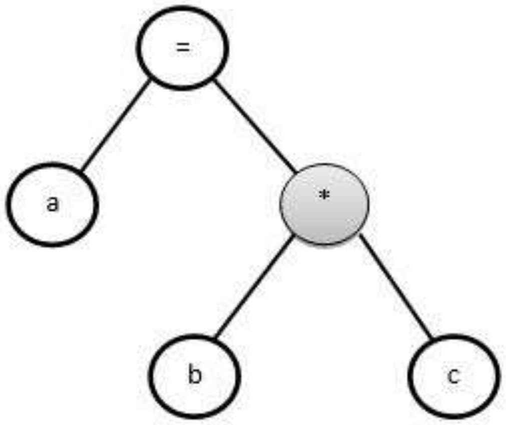

Figure 2.2 Mutated instruction

The original instruction (Figure 2.1) i.e. a = b + c is changed by replacing arithmetic operator '+' to arithmetic operator '*' and the mutated instruction (Figure 2.2) formed is a = b * c. This will generate a mutant program which will be tested by the test cases and if the test case is able to detect this change than the mutant program is said to be killed.

Proposed System

A) New Mutation Operators for java

We are trying to implement some new operators that will help to mutate multithreading programs of java. Some of them are as follows;

STAO	Sleep Time Alteration Operator
IWTO	Inserting Wait Time Operator
ITSO	Insert Thread Synchronization Operator
RTSO	Remove Thread Synchronization Operator

i) STAO: This operator will change the thread sleeping time if sleep time of any thread is mentioned.
ii) IWTO: This operator will insert the waiting time of the thread with some random amount of time.
iii) ITSO: This operator will insert new instruction in the program if there are two or more threads present in the program and there is no synchronization done.

iv) RTSO: This operator will remove thread synchronization of threads in the program if present.

B) Modules and working

There are four modules in this software that has different functions to do and those are as follows.

i) **Scanning:** In this module the software will take a java program as input and it will analyse the program for locations in the source code where the mutation operator can be applied and accordingly it will store a temporary database of implementable mutation operators with the locations in the source code.
ii) **Generating:** The software will read the generated database in scanning phase and will implement all the identified operators on the original program to generate number of mutant programs.
iii) **Testing:** The testing phase includes two sub phases one is testing on original program and second sub phase is testing on mutant programs. These two sub phases are implemented for each test case. The mutation score for each test case is calculated which is in percentage.
iv) **Reporting:** Here the final result will get generated by populating the mutation scores of all the test cases. It also suggest for better test case generation that will help the tester to build better test suite further, Thus increasing the experience of the tester.

The above modules work in a sequence as scanning than generating than testing and then reporting. On identifying the mutation operators from scanning phase, the software will generate all the mutants by applying identified operators to the original program. Here, one mutant will have only one modification from one operator only. Now the test cases will be executed one by one first on original program and then on mutants if successful on original program. If the result of testing on original program and mutant are different that means the test case was able to find out the defect in the mutant program, this kills the mutant indicating success. There are two types of alive mutant, one is that mutant on which the test case failed to identify bug and other is that mutant which cannot be killed at all and those are known as equivalence mutants. Finally the mutation score is calculated for each test case which tells how good the test case is to identify more bugs.

C) Algorithm (pseudo code)

```
START
  Read(Program()) from file
  Read(Moperators()) from Library()
  For each x in Moperators() do
    {
    Analyse(Program(),x)
    If(Analyse()==True) then
      {
      Save(x) in Temp()
      }
    }
  For each x in Temp() do
    {
    Generate(Program(),x) returns M
    Save(M) in Mutants()
    MT++
    }
  Read(Test_Cases()) from Library()
  For each T in Test_Cases() do
    {
    Read(ExpectedOP(),T)
    TRunner(Program(),T) Returns R1()
    S1=Compare(R1(),ExpectedOP())
    If(S1==False) then
      {
      Write(T=OFailed, Report())
```

```
          }
        Else
          {
          For each M in Mutants() do
              {
              TRunner(M,T) returns R2()
              S2=Compare(R1(),R2())
              If(S2==False) then
                  {
                  Mark(M=Killed)
                  MK++
                  }
              Else
                  {
                  Mark(M=Alive)
                  If(FindEM(M)==True) then
                      {
                      ME++
                      }
                  }
              }
        Score=MK/(MT-ME) * 100
        Write(Score, Report())
          }
        }
    Display(Report())
STOP
```

Meaning of Functions used in algorithm:

1. Program(): Original Program to be tested.
2. Library(): Contains all the required operators, functions and Test Cases.
3. Moperators(): Available mutation operators repository.
4. Analyse(): This function checks whether mutation operator can be used or not.
5. Temp(): Repository of all the identified mutation operators.
6. Generate(): This function generates all possible mutant programs.
7. Mutants(): Repository of all generated mutants.
8. Test_Cases(): Repository of Test cases.
9. ExpectedOP(): It is the expected Result when test is ran on original program.
10. TRunner(): This function runs the given test case on given program and returns result..
11. Compare(): Compares two results and returns True is results are same otherwise False.
12. Mark(): Marks the status of Mutant as killed or alive.
13. FindEM(): This function checks whether the mutant is equivalence mutant or not.
14. Write(): Writes the result in the final Report() file.
15. Report(): This is a file that contains complete report of test cases with their scores and also contains what should be done to improve quality of test cases for given program.

D) Architecture

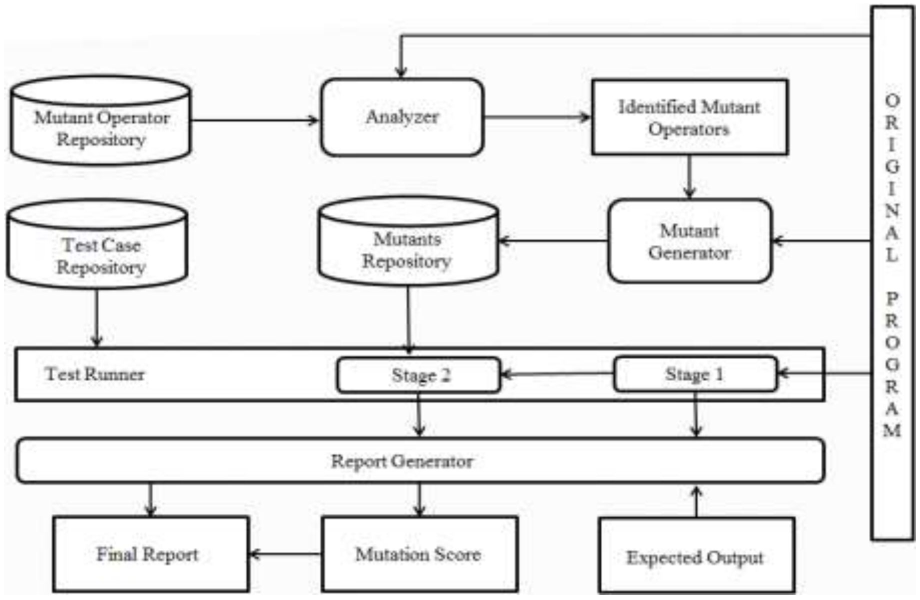

Figure 3.1 Architecture of Mutation Testing

RESULTS AND DISCUSSION

A) Feasibility Analysis

The Feasibility analysis defines in what category the problem comes under. The problems which can be solved in polynomial time are known as P problems and those which cannot be solved in polynomial time are known as NP problems. The problems which can be solved but not in polynomial time are known as NP Complete problem and NP problems whose exact solutions cannot be determined are known as NP Hard problems.

The software in the proposed paper comes under NP Hard category. This is because in mutation testing we cannot determine how many mutants will get generated out of which how many will get killed and what will be the Mutation score. Moreover we cannot determine how long will it take and also it is not possible to guarantee that the quality of test case is good or bad indeed.

B) Mathematical Model

Input

i) Let P be the original java program given as input.
P → Original program
ii) Let T be the Test Suite
T = { ti | ti is a test case }

Output

Let R be the set of tuples as below
R = { T, S, C, R } where,
T → Test Case ID
S → Mutation Score
C → Code Coverage
R → Remark and suggestion

Mutation Score Formula

$$MS = \frac{MK}{MT - ME}$$

Where,

MK = Mutants Killed Count
MT = Total Mutants Count
ME = Equivalence Mutant Count

MS = Mutation Score

Mutant Killing

i) P → Original Program
ii) T → Test case
iii) M → Mutant Program
iv) Let O1 be the result of testing on original program.
TRunner(P,T) returns O1
v) Let O2 be the result of testing on mutant program.
TRunner(M,T) returns O2
(O1 != O2) ? Killed : Alive

CONCLUSION

We have seen what is mutation testing and how it is done. In this paper we have introduced some new mutation operators for java language which are very convenient to implement on java multithreading programs. This covers some missing features of java to be tested at the same time the tester also gets suggestions in how the test suite can be improved.

In future we are planning to develop more mutation operators that can cover even more features of java language which will include java networking, java database connectivity, java servlets and many more. We are also planning to introduce a new service in cloud platform i.e. Mutation Testing as a Service. The testing tool will get deployed on cloud platform and will be accessible to anyone anywhere anytime who wants to test their java source codes.

REFERENCES

[1] Pawar Sujata G. and Idate Sonali R., "Mutation Testing Using Mutation Operators for C# Programs," International Journal of Advanced Research in Computer Science and Software Engineering, Volume 4, Issue 7, July 2014, ISSN: 2277 128X.

[2] Pedro Delgado-P´erez, Inmaculada Medina-Bulo and Juan Jos´e Dom´ınguez-Jim´enez, "Analysis of the Development Process of a Mutation Testing Tool for the C++ Language," Software Engineering Group, UCASE, ICCGI 2014, The Ninth International Multi-Conference on Computing in the Global Information Technology.

[3] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Jozala, "Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation," Transactions on Software Engineering, IEEE, VOL. 40, NO. 1, JANUARY 2014

[4] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," IEEE Transaction on Software Engineering, Oct. 2011, vol. 37, no. 5.

[5] MuJava by Yu-Seung Ma, Je Ofutt, and Yong Rae Kwon , "An Automated Class Mutation System," Division of Computer Science Department of Electrical Engineering and Computer Science Korea, Advanced Institute of Science and Technology, Korea.

[6] Sunita Garhwal, Ajay Kumar and Poonam Sehrawat, Mutation Testing for JAVA, U.I.E.T. Kurukeshtra University, Dept of. Computer Science & Eng. Thapar Institute of Eng. & Tech Lecturer U.I.E.T. K.U.K.

[7] Ali Parsai, "Literature Survey on Mutation Testing," University of Antwerp, Research Internship 2 Report, August 2014.

[8] Manpreet Kaur and Rupinder Singh, "A Review of Software Testing Techniques," International Journal of Electronic and Electrical Engineering, Number 2014 ISSN 0974-2174, Volume 7.

[9] Jinfu Chen, Huanhuan Wang, Rubing Huang, Dave Towey, Chengying Mao, and Yongzhao Zhan, "Worst-Input Mutation Approach toWeb Services Vulnerability Testing Based on SOAP Messages," Volume 19, Number 5, October 2014, ISSNll1007-0214.

ISSN: 2277-9655
(I2OR), Publication Impact Factor: 3.785

AUTHOR BIBLIOGRAPHY

	Prof. Vina M. Lomte is the HOD of Computer Dept. at RMD SSOE College, Pune, having more than 10+ years of experience in the field of teaching and research. The domains of her research are Software Testing, Software Engineering and Web Security.
	Mr. Mitesh Rampariya is currently a student at RMD SSOE College pursuing his BE Degree in the field of Computer Engineering. He is more focused on the algorithm and coding part of Mutation testing tool for java. He is researching to overcome the problem of equivalence mutants too, by finding out some good solution for it.
	Ms. Barkha Sethia is currently a student at RMD SSOE College pursuing his BE Degree in the field of Computer Engineering. She is working on finding more mutation operators that can be implemented in java to mutate programs that cover different features of java, for example java networking, java database connectivity.
	Ms. Vishakha Navandar is currently a student at RMD SSOE College pursuing his BE Degree in the field of Computer Engineering. She is working on developing mathematical model as well as the overall architecture and flow of mutation testing tool.
	Ms. Lubdha Dahale is currently a student at RMD SSOE College pursuing his BE Degree in the field of Computer Engineering. She is working on testing part of this software and finding how the test suite can be improved so that it can achieve code coverage and can fine as many bugs as possible with less number of test cases.