ISSN:0975-9646

Improving the Test Quality of Safety Critical Software by using Combined KPSO Mutation Method.

Dr.S.Preetha
Assistant Professor,
Department of Computer Science,
Sri Ramakrishna College of Arts & Science for Women,
Coimbatore District

Gangarajam,
MPhil Scholar,
Department of Computer Science,
Sri Ramakrishna College of Arts & Science for Women
Coimbatore District

Abstract: Software Testing is the process of executing a program or system with the aim of finding errors. 50% of the total development time is spent on testing the software and correcting them. Tests are commonly generated from program source code, graphical models of software (such as control flow graphs), and specifications / requirements. Testing provides a primary means for assuring software in safety-critical systems. Creating test cases that efficiently checks for faults in software is always a problem. To solve this problem, mutation testing, a fault - based testing technique, used to find the effectiveness of test cases. It is an alternative or complementary method of measuring test sufficiency, achieve the test coverage levels recommended or mandated by safety standards and industry guidelines is applied to ensure the safety criticality and quality of the system. The mutation testing approach proves the test quality by replacing the original contents of the program with mutants generated. In this paper, mutation testing reduces high computational overhead by using every test case to find out the mutants by introducing a new method called combined K-means and Particle Swarm optimization (KPSO) algorithm. It aims to find out the optimal test cases which can predict the changes occurred due to mutants in the program in an efficient manner.

Keywords: Mutants, Faults, KPSO, Test cases, Clustering.

INTRODUCTION:

Software testing is an investigation conducted to provide end-users with information about the quality of the product or service under test. Software testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation. Testing is an essential activity in the verification and
validation of safety-critical software. It provides a level of confidence in the end product based on the coverage of the requirements and code structures achieved by the test cases. It has been suggested that verification and validation require 60 to 70 percent of the total effort in a safety-critical software project.
Software testing can be stated as the process of validating and verifying that a computer program/application/product:

- Meets the requirements that guided its design and development,
- Works as expected,
- Can be implemented with the same characteristics,
- Satisfies the needs of end-users

However, for any other program, faults may occur in any development phase of a software. Testing is fault-based when it seeks to demonstrate that prescribed faults are not in a program. It is assumed that a program can only be incorrect in a limited fashion specified by associating alternate expressions with program expression. Fault-based testing is a software testing methodology using test data designed to demonstrate the absence of a set of prespecified faults; typically, frequently occurring faults. For instance: demonstrate that the software handles or avoids divide by zero correctly, test data would include zero.
Creating test cases that efficiently checks for faults in software is always a problem. To solve this problem, mutation testing, a fault - based testing technique, used to find the effectiveness of test cases. Mutation is a fault-based testing technique, for evaluating, the quality of software .The more efficient the test cases are, the more testing can be performed in a given time.

MUTATION TESTING – AN OVERSIGHT:

Mutation Testing adopts “fault simulation mode”. It has been advocated as a technique for generating test cases by inserting faults in a program and the effectiveness of test suite is represented by ‘mutation score’. Though powerful, mutation testing is computationally expensive, as many mutants need to be produced and executed. The testing technique address the problem of finding a small set of mutation operators and determining the efficiency of high order mutants using fragility values and fitness function, which are sufficient for measuring test effectiveness.
Mutation testing (or Mutation analysis or Program mutation) is used to design new software tests and evaluate the quality of existing software tests. Mutation testing involves modifying a program's source code or byte code in small way. Each mutated version is called a mutant and tests detect and reject mutants by causing the behavior of the original version to differ from the mutant. This is called killing the mutant. Test suites are measured by the percentage of mutants that they kill. New tests can be designed to kill additional mutants. Mutants are based on well-defined mutation operators that either mimic typical programming errors or force the creation of valuable tests. The purpose is to help the tester develop

effective tests or locate weaknesses in the test data used for the program or in sections of the code that are seldom or never accessed during execution.

Mutation testing is done by selecting a set of mutation operators and then applying them to the source program one at a time for each applicable piece of the source code. The result of applying one mutation operator to the program is called a mutant. . If a test cases distinguish the mutant program from the original program in term of output then we say mutant are killed otherwise mutants are alive.

Mutation Score = number of mutants killed / total number of mutants.

For example:

OriginalProgram

```
If ( a && b) {
C = 1;
} else { c=0; }
```

Mutated Program

```
If ( a || b) {
C = 1;
} else { c=0; }
```

Now, for the test to kill this mutant, the following three conditions should be met:

1. A test must reach the mutated statement.
2. Test input data should infect the program state by causing different program states for the mutant and the original program. For example, a test with a = 1 and b = 0 would do this.
3. The incorrect program state (the value of 'c') must propagate to the program's output and be checked by the test.
4. These conditions are collectively called the RIP model.
   - Weak Mutation Testing requires that only the first condition is satisfied. It is closely related to code coverage methods and requires much less computing power.
   - Strong mutation testing requires that both conditions be satisfied. Strong mutation is more powerful, since it ensures that the test suite can really catch the problems.
   - Equivalent Mutants: The resulting program is behaviorally equivalent to the original one. Such mutants are called equivalent mutants.

MUTATION OPERATORS:

Many mutation operators have been explored by researchers. Here are some examples of mutation operators for imperative languages:

- Replace each Boolean sub expression with true and false.
- Replace each arithmetic operation with another, e.g. + with *, - and /.
- Replace each Boolean relation with another, e.g. > with >=, == and <=.
- Replace each variable with another variable declared in the same scope (variable types must be compatible).

These mutation operators are also called traditional mutation operators. There are also mutation operators for object-oriented languages, for concurrent constructions, complex objects like containers, etc. Operators for containers are called class-level mutation operators. Mutation operators have also been developed to perform security vulnerability testing of programs.

LITERATURE SURVEY:

The literature on mutation testing provides an oversight on mutation testing and also discusses various surveys on mutation testing. It also describes the tools, used to build them effectively and helps in reaching a state of maturity and applicability. Mutation testing has contributed a set of approaches, tools, developments and empirical results.

S.NO	YEAR	NAME OF THE AUTHOR	PUBLICATIONS
1	1988	Mathur Krauser,, and Rego	Mutant unification- mutants of the same type be grouped together and that the groups be handled by different processors in the SIMD system
2	1990.	Choi and Mathur	Suggested scheduling mutant executions on the nodes of a hypercube called PMothara.
3	1988.	Hamlet	Embedded in a compiler and performed a version of instrumented weak mutation which is the first. mutation-like testing system.
4	1985	Girgis andWoodward	Implemented a system for Fortran-77 programs that integrates weak mutation and data flow analysis
5	1985	Woodward and Halewood	Introduced the idea of weak and strong mutation.
6	2002	Richardson ,Thompson and Marick	Iimplemented a weak mutation system and reported results from using test data generated strong mutation to find faults that were injected into programs.

S.NO	YEAR	NAME OF THE AUTHOR	PUBLICATIONS
7	2002.	Clark and McDermind, and Chevalley and Thevenod-Foss.Offutt	Developed MuJava-for oo applications for finding faults.
8	1985	P. J. Walsh	Developed a measure of test case completeness
9	2002	Y. S. Ma, Y. R. Kwon	Inter-class mutation operators for Java
10	2005	J. H. Andrews, L. C. Briand, and Y. Labiche	An integrated system for program testing using weak mutation and data flow analysis.
11	2006	James H. Andrews, Lionel C. Briand, Yvan Labiche and Akbar Siami Namin	Test coverage criteria: Block, Decision, C- use and P- use. Useful to assess and compare cost-effectiveness.
12	1993	Ricky W. Butler and George B. Finelli	Driver – to check output for corresponding input. To predict hardware failures and to ensure reliability.
13	2006	Hyunsook Do and Gregg Rothermel	prioritization techniques using mutation faults, focusing on open source Java programs like Junit and TSL to improve fault detection rate.
14	2001	John Joseph Chilenski	Structural coverage -requirements-based verification process.MCDC - verification process executes each side of the sub domain partitions defined by a decision's conditions
15	2006	Lijun Shan and Hong Zhu	It proposes an approach called data mutation to generating a large number of test data from a few seed test cases.

BACKGROUND STUDY:

Testing is used to assure the status of the software that is not in software critical system. In the existing work mutation testing is used to test the software critical level. Mutation testing provides a repeatable process for measuring the effectiveness of test cases and identifying disparities in the test set.

In the existing work, C and Ada coding are taken for checking the mutation process using MILU tool set. The applications of every mutation operator created one or more instances of code item and it is necessary to evaluate the success of each mutant in order to identify mutants whose behavior is identical to original program. Each mutant was recompiled and, assuming that it passed this process, was then run in a simulation environment. In order to reduce the test re-execution time, the tests in this study were run in parallel across numerous dedicated test platforms. Live mutants represented potential improvements to the test-case design. Erroneous behavior may duplicate the local behavior of test cases.

Equivalent mutants add no value to the process since the behavior of these mutants matches the intended implementation and therefore cannot be killed by the test cases. Each instance required a manual review of the test-case set to understand where it failed to kill the mutant. Reviewing the mutants to determine equivalent behavior is overhead and is difficult to automate.

KPSO Methodology and Process:

In this paper, in order to reduce the computational overhead, k means clustering and PSO technology is introduced. K- Means clustering is used to cluster the test cases which are most similar to each other. In this work, the test cases are clustered together based on max values and min values. After finding out the best position i.e., center point of every cluster, the PSO algorithm is applied to every cluster which is used to find the optimal best values i.e. test cases.

The k-Means algorithm is very effective with regard to the computational time or parameter tuning but is applicable to Gaussian clusters of equal volumes. The connectivity principle yields clusters of various shapes but the methods implementing it may suffer from the 'chaining effect' that causes undesirable elongated clusters, or are very sensitive to parameters.

In order to deal with clusters of various shapes, a locality concern may be used:"neighboring" data items should share the same cluster. We propose a Swarm algorithm called PSO-kMeans which implements this simple connectivity principle and introduces it within k-Means, taking thus into account simultaneously the local and the global distribution in data.

Particle swarm optimization (PSO) is a computational method that optimizes a problem by iteratively trying to improve a candidate solution with regard to a given measure of quality. PSO optimizes a problem by having a population of candidate solutions, here dubbed particles, and moving these particles around in the search-space according to simple mathematical formulae over the particle's position and velocity.

Each particle's movement is influenced by its local best known position and is also guided toward the best known positions in the search-space, which are updated as better positions are found by other particles. This is expected to move the swarm toward the best solutions. In this work, PSO finds the best value of the each candidate the algorithm found the best location for changing the velocity of the each particle.

The K-means algorithm is a fast method due to its simple and small number of iterations. But the dependency of the algorithm on the initialization of the centers has been a major problem and it usually gets stuck in local optima though it tends to converge faster than the PSO algorithm. Using the merits of both algorithms, PSO and K-means are combined.

Test Case Generation

- The possible test cases (inputs) will be generated for finding the criticality of the software and to compare the result of the programs without mutation and after mutation.
- This test case generation can produce two types of results. Those are survived mutants and the kill mutants.
- Survived mutants are the mutants who cannot be find out after applying test cases.
- Kill mutants are the one which will be identified and deleted by the test cases.
- Thus the test cases will be generated to reduce the survived mutants in order to improve the software functionality.

Clustering test cases

The K-means algorithm groups the set of data points in space into a predefined number of clusters and Euclidean distance is commonly used as a similarity measure.

- Place K points into the space represented by the objects that are being clustered. These points represent the initial group is reached.
- Calculate the distance between the cluster centre and the data vectors according to the eq.,

$$d(X_i, C_i) = \left\{ \sum_{i=1}^{dw} (X_{im} - C_{jm}) \right\}^{1/2} \qquad (1)$$

s the minimum distance.

- When all the objects have been assigned recalculate the cluster center according to eq.,

$$C_j = \frac{1}{n} \sum_{x \in s} X_j \qquad (2)$$

Finding optimal test cases

- The K-means algorithm is a fast method due to its simple and small number of iterations.
- But the dependency of the algorithm on the initialization of the centers has been a major problem and it usually gets stuck in local optima though it tends to converge faster than the PSO algorithm.
- PSO clustering algorithm performs a global search in the entire solution space.
- Using the merits of both algorithms, PSO and K-means are combined.
- In the new algorithm a single particle represents a set of cluster centers, that is, a particle represents one possible solution for clustering and the position of each particle xi is constructed as,

$$xi = (c_{i1}, c_{i2}, \ldots, c_{iK})$$

Where, K is the number of clusters, cij is the j-th cluster centre of the i-th particle. Then the swarm represents a candidate cluster result. The fitness of each particle is measured as,

$$\text{Fitness} = 1 \Big/ \sum^{K} \sum d(x_i, c_j)$$

Where d(xi, cj) is defined in the Equation and cj is the j-th cluster

Mutation generation

The results obtained in the clustering and PSO techniques are then applied in mutation testing to find efficient test cases which are able to distinguish mutant program from original program. Mutation testing involves the substitution of simple code constructs and operands with syntactically legal constructs to simulate fault scenarios. The method level and class level operators are selected and executed.

The mutated program, i.e., the mutant, can then be re-executed against the original test cases to determine whether it can kill the mutant exists (i.e., killed mutants exhibit different behavior from the original program when exercised by one or more test cases). If the mutant is not killed by the test cases, then these test cases are insufficient and should be enhanced. This process of re-executing tests can continue until all of the generated mutants are captured (or “killed”) by the test cases.

RESULTS AND DISCUSSIONS

The purpose of the experiment is to find efficient test cases using mutation testing together with KPSO techniques. The result of the existing work is compared with the proposed work. Therefore, the performance of the system is evaluated based on the following performance metrics.

Time Performance:

The time quantifies the amount of time taken by an algorithm to run as a function of size of the input to the problem. The following graph fig 5.1 indicates the performance measures for the time complexity. It indicates the total time taken to find out the mutants applied in the program.

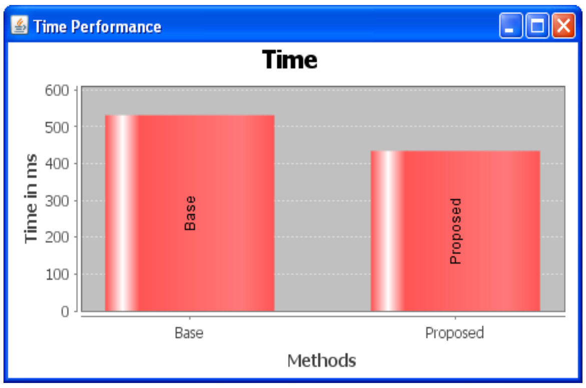

Fig :1 Time performance

Memory Performance:

The memory quantifies the amount of memory taken by an algorithm to run as a function of size of the input to the problem. The following graph fig 5.2 shows the memory utilization level. The result showed is memory utilized while trying to find out the mutants applied on the original coding.

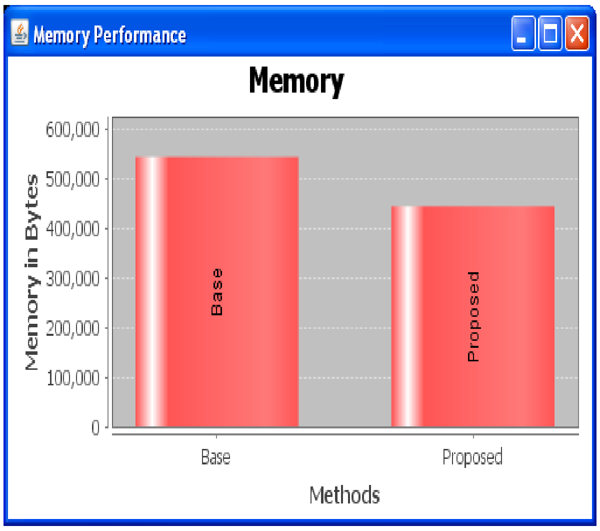

Fig :2 Memory performance

Method Level Mutation Values:

Method mutation operators are developed to handle all the possible syntactic changes in the methods incorpate in the program. It is stable and minimizes the number of equivalent mutants that they generate.

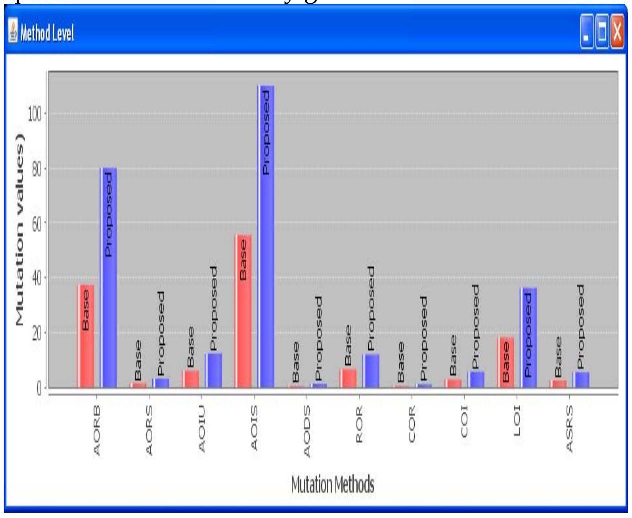

Fig :3 Method level mutation values

Class Level Mutation Values:

Class mutation operators introduce faults into classes defined in the program using a set of class mutation operators to handle syntactic modifications in the program. The following graph fig 5.4 indicates the number of mutation that is found while replacing the code with mutants in the entire class of the programs.

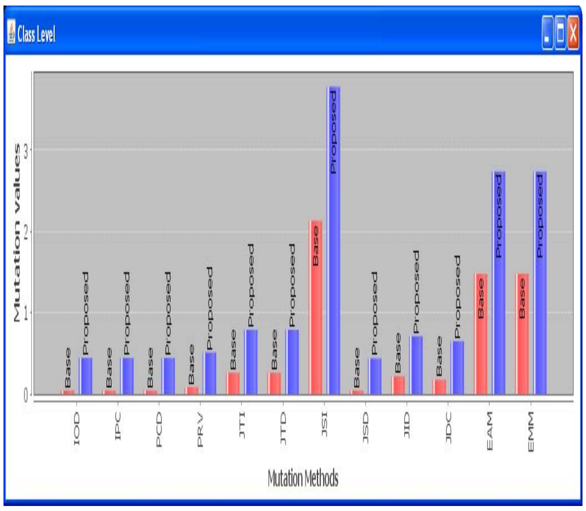

Fig :4 Class level mutation values

CONCLUSION AND FUTURE WORK:

In this work, the efficient test cases are identified which can identify the created mutants with the high flexibility. In this work, k-means algorithm is used to identify the best test cases which can identify the mutants with the most probability by grouping the similar test cases. The proposed k-means based particle swarm optimization technique is used to find the test cases which are most efficient to predict the mutants generated in the coding.

The performance analysis of our work proves that the proposed method is more efficient than the existing work. It can able to find the mutants generated manually with less time complexity, less memory efficiency and more number of killed mutants.

In future, the high dimensional problems (programs) ad large number of patterns can be handled. To achieve this, the fitness function calculation can be extended. By extending the fitness function, explicit optimization of the inter- and intra-cluster distances can supported. The PSO clustering algorithms will also be extended to dynamically determine the optimal number of clusters.

REFERENCES:

[1] J. Offutt and R. H. Untch, “Mutation 2000: Uniting the Orthogonal,” in Proceedings of the 1st Workshop on Mutation Analysis (MUTATION’ 00), published in book form, as Mutation Testing for the New Century. San Jose, California, 6-7 October 2001, pp. 34-44.

[2] K. N. King and A. J. Offutt, “A Fortran Language System for Mutation- Based Software Testing,” Software:Practice and Experience, vol. 21, no. 7, pp. 685–718, October 1991

[3] T. A. Budd and D. Angluin, “Two Notions of Correctness and Their Relation to Testing,” Acta Informatica, vol. 18, no. 1, pp. 31–45, March 1982.

[4] P. J. Walsh. A measure of test case completeness (software,engineering). PhD thesis, State University of New York at Binghamton,Binghamton, NY, USA, 1985

[5] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses versus mutation testing: An experimental comparison of effectiveness. Journal of Systems and Software, 38:235–253, 1997

[6] J. Choi and A. P. Mathur. Use of fifth generation computers for high performance reliable software testing. Technical report SERC-TR-72- P, Software Engineering Research Center, Purdue University, West Lafayette IN, April 1990.

[7] E. W. Krauser, A. P. Mathur, and V. Rego. High performance testing on SIMD machines. In Proceedings of the Second Workshop on Software.

[8] Aditya P. Mathur and E. W. Krauser. Modeling mutation on a vector processor. In Proceedings of the 10th International Conference on Software Engineering, pages 154{161, Singapore, April 1988. IEEE Computer Society Press.

[9] Aditya P. Mathur and E. W. Krauser. Mutant unification for improved vectorization. Technical report SERC-TR-14-P, Software Engineering Research Center, Purdue University, West Lafayette IN, April 1988.

[10] M. R. Girgis and M. R. Woodward. An integrated system for program testing using weak mutation and data flow analysis. In Proceedings of the Eighth International Conference on Software Engineering, pages. 313{319, London UK, August 1985.

[11] R. G. Hamlet. Testing programs with the aid of a compiler. IEEE Transactions on Software.

[12] R. A. DeMillo and E. H. Spafford, `The Mothra software testing environment', Proceedings of the 11th NASA Software Engineering Laboratory Workshop, Goddard Space Center, December 1986.

[13] R. A. DeMillo, E.W. Krauser, R. J. Martin, A. J. Offutt and E. H. Spafford, `The Mothra tool set', Proceedings of the Hawaii International Conference on System Sciences, Kailua-Kona, HI, January 1989.

[14] Offutt, `An extended overview of the Mothra software testing environment', Proceedings of the IEEE Second Workshop on Software Testing, Verification and Analysis, Banff Alberta, July 1988.

[15] L. J. Morell. A Theory of Error-Based Testing. PhD thesis, University of Maryland, College Park MD, 1984. Technical Report TR-1395.