Journal of Computer Science 7 (10): 1581-1589, 2011
ISSN 1549-3636

A Tool for Generation and Minimization of Test Suite by Mutant Gene Algorithm

[1]Selvakumar Subramanian and [2]Ramaraj Natarajan
[1]Department of Information Technology,
Thiagarajar College of Engineering, Madurai, Tamil Nadu, India
[2]Department of Computer Science and Engineering,
G.K.M College of Engineering,
Chennai, Tamil Nadu, India

Abstract: Problem statement: This study proposes a new idea for generation of minimized test suite in the test case generation using the mutant gene algorithm, which not only identifies the best test cases but also reduces the number of test cases generated, selects test cases optimally there improving the performance in testing of software. Test cases are generated by using branch coverage algorithm and a coverage table is created for verifying branch coverage. **Approach:** The process of minimization was done through Mutant gene algorithm. Mutant gene algorithm combined both the mutation testing process and genetic algorithm. Initially a number of chromosomes were generated in random order. Mutation score was used for finding fitness function. The fitness function was found for all the randomly generated chromosomes by applying the mutant score to the function. Rank based selection was used for selecting the chromosomes. After the selection of the chromosomes one-point crossover was performed. A population of chromosomes obtained, which was given as the input for the next iteration. Large iterations were performed to obtain the best test case with higher fitness value, it was the end condition. **Results:** Between the measured iterations the value of the mutant score remained constant. The results of the experiments showed that the minimization process was competitive with other methods and even outperforms them for complex cases. **Conclusion:** The whole generation and minimization process was fully automated; redundant explorations of test case were avoided, resulting in efficient generation of test cases.

Key words: Test case generation, test suite minimization, test suite reduction, genetic algorithms, test data, mutation testing, mutants detected, System Under Test (SUT), mutant gene algorithm, redundant explorations

INTRODUCTION

Testing of software is the appendage used to assess the quality of computer software. Software testing is an empirical proficient investigation conducted to provide stakeholders with information about the quality of the product or service under test, with respect to the context in which it is intended to function. The testing of software is an important means of assessing the software to determine its quality (Mohammad, 2008). Software testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its requirements (Mohammad *et al*., 2010). Software testing includes the process of executing a program or application with the intent of finding software bugs. The complexity of software systems has been increasing dramatically in the past decade and software testing as a labor-intensive component is becoming more and more expensive. As software develops and evolves, new test cases are continually generated to validate the latest modifications. As a result, the sizes of test suites grow over time (Lin *et al*., 2008). Testing costs often account for up to 50% of the total expense of software development; hence any techniques leading to the automatic generation of test case will have great potential to considerably reduce costs. Current approaches to verification in the software industry in general are losing the battle. While systems grow evermore complex, the percentage of total costs consumed by verification grows and grows, as defect rates increase. Currently, over half of all errors are not

Corresponding Author: S. Selvakumar, Department of Information Technology, Thiagarajar College of Engineering, Madurai, Tamil Nadu, India

found until 'down-stream' in the development process or during post sale software use. While in practice test cases are often generated manually, there has been a great deal of research on techniques for automated test case generation. These massive budgetary impacts suggest that current approaches are failing and new approaches must be investigated with great haste. An aims to improve an automated test case generation method to minimize a number of test cases while maximizing an ability to identify critical domain specific requirements (Kosindrdecha and Daengdej, 2010). To use the test automation tools is one of good approaches. They are skillfully devised for convenient use and give testers the testing environment which is more correct than the manual testing. Helping engineers create test cases is important, but test cases are useful only if they can reveal faults. To reveal faults, test cases must produce observable failures.

Test data generation uses the branch coverage algorithm to select a path that may reach the targeted branch and obtains constraint information for the selected path and to generate the test cases. After the constraints have been collected from the program with the assistance of a branch coverage algorithm we generate the test data inputs and update the values in coverage Table to ensure that the selected branch is reached and traversed. The coverage Table, Michael's approach is established to record the branch information of the program under test and keeps track of whether a branch is tested or not. Mutation testing is a method of software testing, which regards modifying programs source code or byte code in small ways. Any tests which pass after code has been mutated are considered defective, called as mutations, are established on well-defined mutation operators that either mimic typical software error 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 parts of the code that are seldom or never accessed during execution. Genetic algorithm states about the ability of simple representations of bit strings to encode complicated structures and the power of simple transformations to improve such structures. In order to solve a problem, genetic algorithm, requires a chromosomal representation of solution to the problem, a way to create an initial population of solutions, an evaluation function that plays the role of the environment, quality rating for solutions in terms of their "fitness", Genetic operators that alter the structure of "children" during reproduction, values for the parameters that genetic algorithm uses (population size, probabilities of applying genetic operators).

Related work: In the literature, almost all the approaches to test case generation will consider how to avoid generating redundant test cases (Yanping *et al*., 2009; Dmitry *et al*., 2010; Alshraideh, 2008; Razali and Garratt, 2010). Piles of efforts have also been put into research on how to reduce the test suite size of a previously acquired test suite while maintaining its effectiveness. (Yanjun *et al*., 2010) recommends Greedy algorithm and with the growth of test size and suggests the usage of greedy evolution and GRE for more general by analyzing the influencing factors and performance the running time of algorithms of distinctions of 6 classical algorithms viz greedy, greedy evolution, heuristics, GRE , ILP and GA. A method for test suite minimization that uses an additional testing criterion to break the ties in the minimization process, under specific conditions, their proposed approach can also accelerate the process of minimization. The work proposed by (Yanping *et al*., 2009) is a model-based regression test suite reduction method that considers an SDL model representing the requirements of a System Under Test (SUT) and a set of modifications on this model and reduces the size of a given regression test suite by examining interaction patterns covered by the test suite. (Dmitry *et al*., 2010) performs experimental evaluation for the use for test suites reduction for software integration testing.

Existing approaches of automatic test data generation have achieved some success by using evolutionary computation algorithms, but they are unable to deal with Boolean variables or enumerated types and they need to be improved in many other aspects. The major drawback in the earlier works is that only test data generation is performed and the test cases are derived without minimization of test cases. The level of confidence in a software component is often linked to the quality of its test cases. This quality can in turn be evaluated with mutation analysis: faulty components (mutants) are systematically generated to check the proportion of mutants detected ("killed") by the test cases. But while the generation of basic test cases set is easy, improving its quality may require prohibitive effort work focuses on the issue of automating the test optimization. Our proposed work which looks at genetic algorithms to solve this problem and model it as follows: a test case can be considered as a predator while a mutant program is analogous to a prey. The aim of the selection process is to generate test cases able to kill as many mutants as possible. But neither the effectiveness of the test case nor the fault detection loss of reducing the generated test cases by selecting optimal test cases are not dealt. Genetic algorithms are very efficient in the problems of exploration and seem to be able to find the single

optimal solution in a huge space of possible solutions (Yedjour *et al*., 2010). (Nazif and Lee, 2010) considered the application of a genetic algorithm to vehicle routing problem. Mao (2010) proposed formal semantics of ontology to improve genetic algorithm in several aspects and make it more adaptive to solve semantic-based problems. The proposed tool addresses this issue, specifically the test case generation and then we select minimized test cases using genetic algorithm.

MATERIALS AND METHODS

Our proposed work is to select the optimal test cases which are effective and the number of generated test cases is also minimum there by reducing the cost of generation of test cases. Overall process as follows:

Step 1: Generate mutant programs by changing >,<,=,!= in the subject program
Step 2: For the generated test case input apply mutant programs
Step 3: Compare the results of original result and the mutant results
Step 4: For each test case input i.e. Test case id find the number of mutant programs is identified
Step 5: Optimal test cases are found using mutant gene algorithm

Initially we generated test case inputs through combinations which reduce the manual work, because instead of giving every input it's adequate, if the values for n and r are given. Where n stands for test data inputs and r denotes subset size:

C=n! / (r!*(n-r)!)

A coverage table is established, showing the predicate number, program line numbers of predicates, predicate, true/false branch and branch coverage status. Before starting to generate test data for the tested program, a seed input, generated randomly is used to execute the program under test for the first time. Generally, running the program with the seed input will result in some branches being tested. The result of the execution of the tested program with the seed input is recorded in the coverage table and the status of each branch is initialized. After the initial coverage table has been established, the next issue is which branch is our next target. Since traversing of different branches will have different contributions to the satisfaction of our selected criterion, we need to weight the importance of each branch towards the branch coverage criterion. We give higher priority to those branches that their traversing causes additional branches to be traversed or makes other branches easier to be traversed. In the proposed approach, we try to discriminate branches and decide their possible contributions to our software testing adequacy criterion, to always work on the highest 'value' branch first.

Mutation testing: Mutation testing is a technique which was first designed to create effective test data, with an important fault revealing. The process of mutation testing is, firs start with a piece of development code that's comfortably covered by unit tests. Once it's verified that all tests pass for a given piece of code it's time to apply the mutation to the target assembly. The extent of the mutation that is applied to the code can span many levels; some of the more coarse mutations merely involve substituting a logical operator with its inverse. For instance, == can turn != while < can turn >=. In more complex mutations it may go so far as to make over the order of execution of code or even remove some lines of code completely. However, as mutations of this degree can often cause compiler errors, it's often easier to initially stick with the simpler mutations mentioned. After the code is mutated, original suite of unit tests is re-run against it. If the tests are well written, any test that covers the mutated program code should fail. However, if the tests succeed in spite of the mutated program code then the tests creates false positives and need to be revisited. If the tests are well written, any test that covers the mutated program code should fail. However, if the tests succeed in spite of the mutated source code then the tests are creating false positives and need to be revisited.

Mutation score: The advantage of the mutation score is that even if no fault is found, it still measures how well the software has been tested giving the user information about the program test quality. On the test selection process, a mutant program is said to be killed if at least one test case discovers the fault injected into the mutant. Conversely, a mutant is said to be alive if no test cases detect the injected fault. Let d be the number of dead mutants after applying the test cases, m the total number of mutants and equiv, the number of equivalent mutants. The mutation score MS for test cases set T is defined as follows:

$$MS(T) = 100\left(\frac{d}{m - equiv}\right)$$

Architectural approach: Conceptual models specify the characteristics of the existing and future systems. They are mainly produced through the use of a designated modeling notation (Rozilawati and Paul, 2010). The architecture that is outlined describes interfaces and behavioral requirements for the minimization of the suite size.

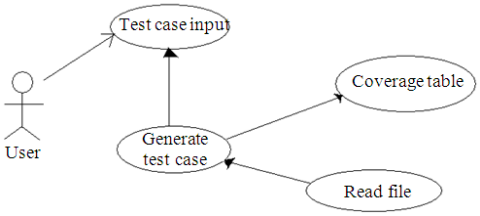

Fig. 1: This use case diagram describes the generation of test cases

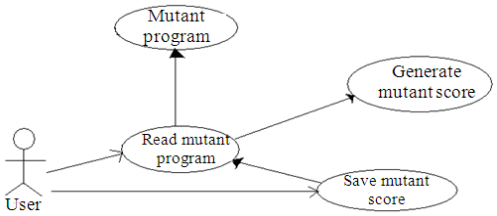

Fig. 2: This use case describes the mutant score generation.

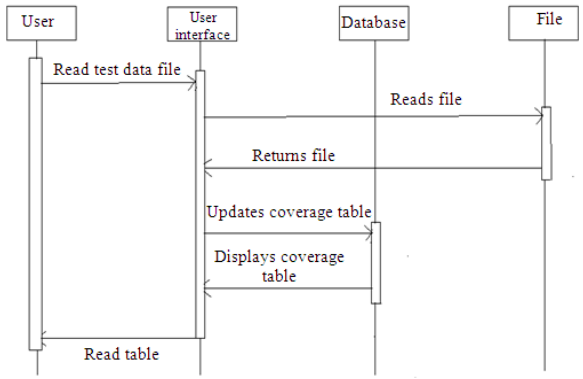

Fig. 3: A sequence diagram showing the sequences for coverage table generation

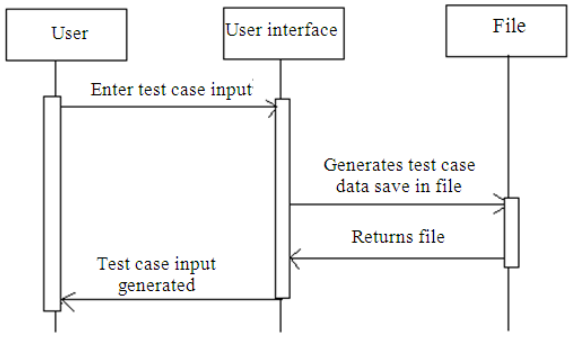

Fig. 4: A sequence diagram showing the various sequences for the test case generation

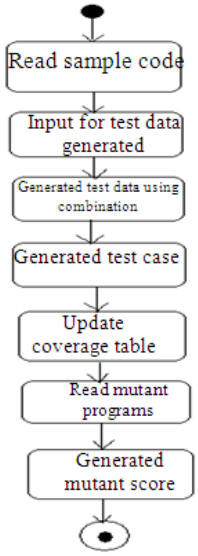

Fig. 5: Activity diagram showing the activities that occur in the generation of mutant

We implemented and validated many of the ideas that were discussed in the system. Figure 1-5 shows the modeling of the tool for Generation and Minimization of Test Suite with Mutant Gene Algorithm.

The following discussion, points the mutation operators that were used in the experimentation. This choice has been channelized by the specific use of mutation analysis for test cases at system level. The discussion also describes the general test selection process based on mutation analysis and pinpoints which part of the process need to be automated.

Mutation analysis for testing the software: Mutation analysis serves the tester create test data and then interacts with the tester to ameliorate the quality of that test data. Mutation analysis involves constructing a set of mutants of the test source, each of which is a version of the test program that differs from the original by one mutation. A mutation is a single syntactic change that is made to a program statement. In an object-oriented context, the class is frequently considered as the unit for testing and mutation analysis has been successfully used to guide the generation of test cases for a class. When applying mutation analysis for system testing, scale problems appear. In the following, we call a mutant program, a software system in which an error has been injected. A system is composed of several classes and each of them can generate many mutants (many faults can be injected). For example, a large number of operators is used which generate large sets

of mutants that are necessary to have a precise evaluation of test cases for one class. The number of mutant programs thus increases with the size of the SUT. Moreover, since all the test cases must be executed against all the mutants, the execution time increases with the number of mutants. Mutation analysis at system level can thus become very time-consuming. At last, if mutant equivalence is often decidable on a class, it is not possible for a tester to decide system equivalence.

The solution chosen is to select two mutation operators to avoid generating too much mutant programs. This subset of operators is still efficient since we expect classes to be tested at unitary level (so all operators have been applied on the code separately). System testing then focuses on the relationships between the classes in the system. Since the purpose of unit and system testing is different, mutation analysis also has to have a different role. The functionality of mutation operators are as described below:

EHF: Causes an exception when executed. This operator allows forcing code coverage.
AOR: Replaces occurrences of "+" by "-" and vice-versa.
LOR: Each occurrence of one of the logical operators (and or, nand, nor, xor) is replaced by each of the other operators; in addition, the expression is replaced by TRUE and FALSE.
ROR: Each occurrence of one of the relational operators (<, >, <=, >=, =, /=) is replaced by each one of the other operators.
NOR: Replaces each statement by the Null statement.
VCP: Constants and variables values are slightly modified to emulate domain perturbation testing. Each constant or variable of arithmetic type is both incremented by one and decremented by one. Each boolean is replaced by its complement.
MCP: Methods calls are replaced by a call to another method with the same signature.
RFI: Stuck-at void the reference of an object after its creation. Suppress a clone or copy instruction. Insert a clone instruction for each reference affectation.

Mutant gene algorithm is applied to identify the test cases which are optimal based on the mutation score of each test case.

Gene modeling for test minimization: For the problem of test minimization, a gene is modeled as a test case (Yanjun *et al*., 2010). In the particular case of a parser a gene is a source file for the particular language. Each file contains several constructs from the language (nodes from the syntactic tree). If there are x nodes in the file a gene can be represented as follows:

$G = [N_1,\ldots,N_x]$.

Another aspect of the algorithm is that, has to be decided for the particular problem of test minimization: the fitness function. We have chosen the mutation score of an individual as the fitness function

Fitness function: The fitness values for an individual list its associated mutation scores. An individual is a set of genes. Let $I=[G_1, \ldots, G_n]$ be an individual composed of *n* genes. Let S_i be the set of mutants detected by G_i. Let *nbMutants* be the total number of mutants generated for the component under test. The fitness function of individual I is computed (Yanjun *et al*., 2010) as:

$$F(I) = (\text{car}\left(\sum_{i=1}^{m} US_i\right) / \text{nbMu tan ts}) \times 100$$

The union set of all S_i corresponds to the set of mutants killed by the individual. The cardinal of this union is thus the number of mutants killed by the individual (Yanjun *et al*., 2010). Then the mutation score of the individual is the percentage of the global set of mutants it can kill. Now, let us define the genetic operators for the particular problem of test suite minimization.

Reproduction: the slot for each individual in the roulette wheel is proportional to its mutation score (Yanjun *et al*., 2010).

Crossover: Crossover is a genetic operator used to vary the programming of a chromosome or chromosomes from one generation to the next. It is analogous to reproduction and biological crossover, upon which genetic algorithms are based (Yanjun *et al*., 2010).

Mutation: Based on the gene modeling, the mutation operator consists in replacing a syntactic node in a source file (an individual) by another licit node (Yanjun *et al*., 2010).

The mutation operator thus chooses a gene at random in an individual and replaces a node in that gene by another one as illustrated.

$G = [N_1, N_i,\ldots N_x]$

$Gmut = [N_1, N_{imut}, \ldots N_x]$.

Table 1: Test Case mutant score

Test case Id	No. of mutants detected
1	20
2	30
3	45
4	35
5	40
6	45
7	30
8	25
9	45
10	40
11	18
12	30
13	45
14	50
15	20
16	45
17	25
18	30
19	15
20	40

Table 2: Coverage table, before applying the branch coverage algorithm for the triangle program

id	Coverage predicate	branch	Status
1	if((i<=0)||(j<=0)||(k<=0))	true	u
2	if (i == j)	true	u
3	if (i == k)	true	u
4	if(j==k)	true	u
5	if(tri==0)	true	u
6	if((i+j<=k)||(j+k<=i)||(i+k<=j))	true	u
7	if(tri>3)	true	u
8	if((tri==1)&&(i+j>k))	true	u
9	if((tri==2)&&(i+k>j))	true	u
10	if((tri==3)&&(j+k>i))	true	u
11	if((i<=0)||(j<=0)||(k<=0))	false	u
21	if (i == j)	false	u
31	if (i == k)	false	u
41	if(j==k)	false	u
51	if(tri==0)	false	u
61	if((i+j<=k)||(j+k<=i)||(i+k<=j))	false	u
71	if(tri>3)	false	u
81	if((tri==1)&&(i+j>k))	false	u
91	if((tri==2)&&(i+k>j))	false	u
101	if((tri==3)&&(j+k>i))	false	u

Table 3: Coverage table after applying the branch coverage algorithm

id	Coverage predicate	branch	Status
1	if((i<=0)||(j<=0)||(k<=0))	true	u
2	if (i == j)	true	u
3	if (i == k)	true	c
4	if(j==k)	true	c
5	if(tri==0)	true	u
6	if((i+j<=k)||(j+k<=i)||(i+k<=j))	true	u
7	if(tri>3)	true	c
8	if((tri==1)&&(i+j>k))	true	u
9	if((tri==2)&&(i+k>j))	true	c
10	if((tri==3)&&(j+k>i))	true	c
11	if((i<=0)||(j<=0)||(k<=0))	false	u
21	if (i == j)	false	c
31	if (i == k)	false	u
41	if(j==k)	false	c
51	if(tri==0)	false	c
61	if((i+j<=k)||(j+k<=i)||(i+k<=j))	false	u
71	if(tri>3)	false	u
81	if((tri==1)&&(i+j>k))	false	c
91	if((tri==2)&&(i+k>j))	false	c
101	if((tri==3)&&(j+k>i))	false	c

A novel method for stopping the iteration is proposed, where the optimal test cases generated is limited between half the number of test cases which was initially selected (for maximum limit) and Square root of number of test cases which was initially selected (for minimum limit).

An illustrative example: The following example is for triangle program, which is one the subject program utilized in the experimentation. Table 1 shows the values of test case mutant scores, test case ids and the corresponding number of mutants detected. The Table 2 shows the coverage table, before applying the branch coverage algorithm for the triangle program. The status for all the constraints is set to untested. The Table 3 depicts coverage table after applying the branch coverage algorithm. The branches (constraints) which are tested are updated as checked.

RESULTS

All the implemented techniques were executed on a PC with an Intel Pentium Dual CPU T3400 @ 2.16 GHz 2.17 GHz CPU and 2 GB memory running the Windows 2000 Professional operating system. The studied test suite minimization techniques were implemented by the students of Information Technology using Microsoft visual studio 2005. The platform chosen for implementing is Visual C# and is chosen for some of its advantages like, better performance of some functions, such as those that run mathematical operations such as combinations and permutations might perform better when they are compiled assemblies that are built from a Visual C# project. Visual C# provides capabilities such as arrays, sophisticated exception handling and reusability of code. Figure 6 shows the interface which describes the overall functionality for automatic test case generation. The Fig. 7 shows the source code, which is the code which is the SUT for which the test case is to be generated. Figure 8 shows the generation of test data. For better understanding the illustrative example described in the earlier is taken as the source. Figure 9 shows the result of test data generation where the test data are generated using combinations and all the test data are saved in a File. Figure 10 shows the Coverage Table, in the interface the 'Generate test cases' button is to generate test cases after reading the test data which was previously stored.

Then the coverage table is created. The test data are generated when all the status of the branch in the coverage table values are 'c' checked. Figure 11 shows the mutant score generation is this process the mutant programs are taken and the test cases are generated for the same test data, the results are stored in text file.

Fig. 6: Interface for automatic test case generation

Fig. 7: Read sample code

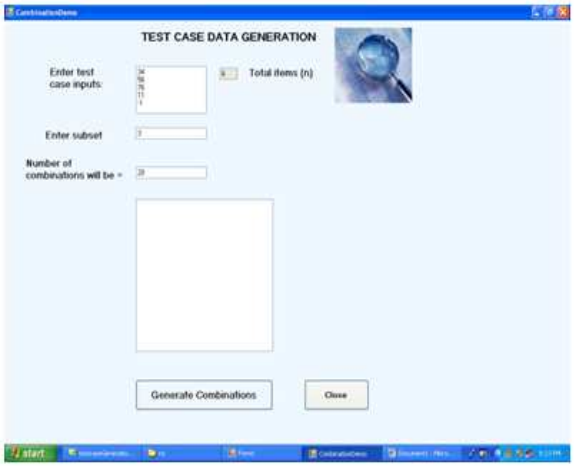

Fig. 8: Test data generation

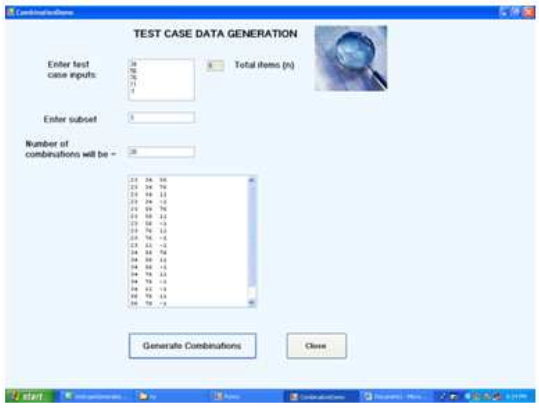

Fig. 9: Result of test data generation

Fig. 10: Coverage table

Fig. 11: Mutant score generation

Table 4: Experimentation 1: Academic subject programs

Name	Lines of code	No. of Classes
Triangle	123	2
Sample	66	1
Average	131	1
Greatest number	186	1
Gcd	142	2

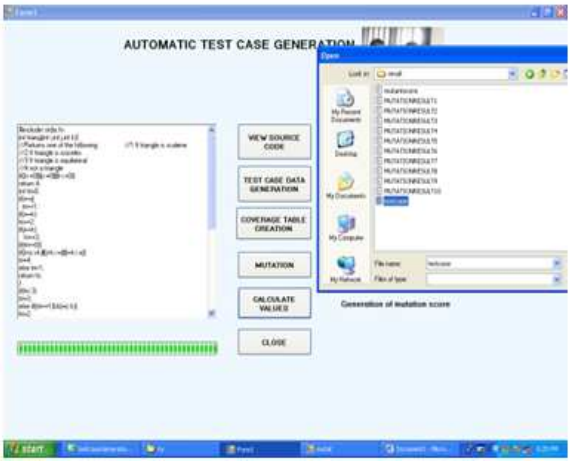

Fig. 12: Calculate Mutant score

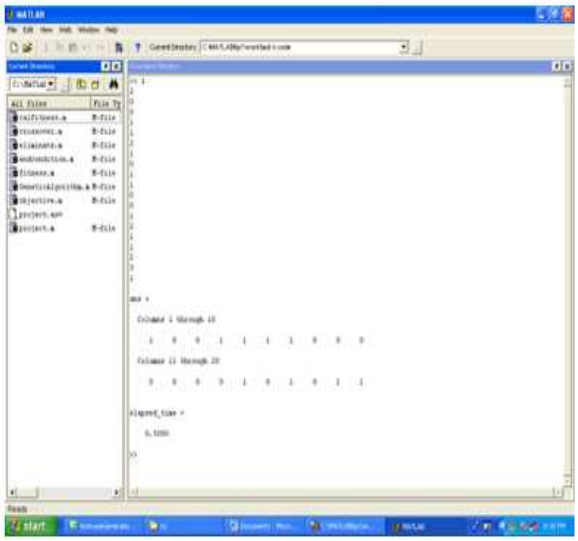

Fig. 13: Final optimized result

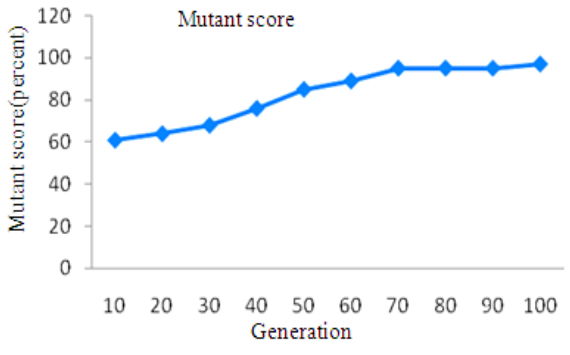

Fig. 14:mutants detected for the corresponding subject programs

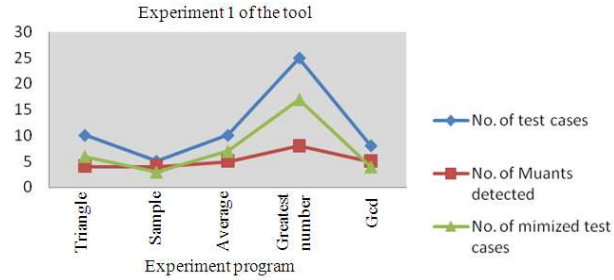

Fig. 15:Results of experimentation on academic subject programs

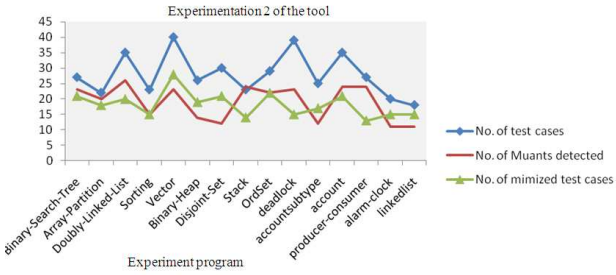

Fig. 16: Results of experimentation of SIR objects

Figure 12: depicts the interface to calculate the mutant score where the Mutant score is generated for the generated test cases. Figure 13 shows the final optimized result.

Subject programs, faulty versions and test case pools: The programs described in Table 4-5 were used as the subject programs. These objects that are retrieved from the Software Infrastructure Repository (SIR 2010) and the programs developed as an academic project by students described in Table 4 were also experimented.

Figure 14 shows the detected mutants for the corresponding subject programs, depicts the increase in the mutant score as the number of iterations in GA increase. Between the measured iterations the value of the mutant score remains constant. The results of the experiments show that the minimization process is competitive with other methods and even outperforms them for complex cases. Even though the other methodology yields a covering test case faster than this mutant gene tool in some cases, the latter is much faster than the other methodology in the majority of the cases. Figure 15 shows the results of experimentation on academic subject programs and Fig. 16 shows the results of experimentation of SIR objects.

Table 5: Experimentation 2: Objects from Software Infrastructure Repository (SIR, 2010)

Name	Lines of code	No. of classes
Binary-search-tree	130	3
Array-partition	13	1
Doubly-linked-list	277	1
Sorting	130	1
Vector	254	1
Binary-heap	72	2
Disjoint-set	35	1
Stack	114	5
Elevator	934	12
OrdSet	229	2
deadlock	24	4
accountsubtype	89	6
Account	66	3
Producer-consumer	99	8
Alarm-clock	125	6
linkedlist	121	5

DISCUSSION

This result indicates that the tool is an attractive alternative since it is just as good as or even better than some of the existing tool in terms of effectiveness and efficiency and is a much simpler process with significantly fewer parameters that need to be adjusted by the tester. However, more experiments with further test objects taken from various application domains must be carried out in order to be able to make more general statements about the relative performance of the proposed tool for test case generation and minimization.

CONCLUSION

In this study a new test suite generation and minimization tool based on mutant gene algorithm is proposed. The proposed method manifested that this test suite minimization results in more optimal than other test suite minimization techniques and also the coverage is improved. All other algorithms concentrate only on best test case selection, but the proposed method selects test cases optimally there improving the performance in testing of software. The proposed approach has the several advantages, the whole generation and minimization process is fully automated; redundant explorations of test case are avoided, resulting in efficient generation of test cases, But then test data generation and minimization still possesses limited capabilities when compared to the requirements of an industrial strength automatic test generation engine. The proposed method could be extended towards the handling of the large software application. Also can be extended for test management and product line approaches and it can also be extended as metrics to assess test case design quality. Although only branch-type coverage measures are chosen as the test adequacy criteria, the new approach can also be extended to other test criteria, such as path coverage.

ACKNOWLEDGMENT

We thank Dr. Gregg Rothermel, Department of Computer Science, University of Nebraska for providing the Siemens Suite of programs and SIR objects.

REFERENCES

Kichigin, D., 2010. A method for test suite reduction for regression testing of interactions between software modules. Lecture Notes Comput. Sci.,. 5947: 177-184, DOI: 10.1007/978-3-642-11486-1_15

Kosindrdecha, N. and J. Daengdej, 2010. A test case generation process and technique. J. Software Eng., 4: 265-287. DOI: 10.3923/jse.2010.265.287

Lin, J.W. and C. Y. Huang, 2009. Analysis of test suite reduction with enhanced tie-breaking techniques. Inf. Software Technol., 51: 679-690. DOI: 10.1016/j.infsof.2008.11.004

Mohammad F.J. Klaib, Sangeetha Muthuraman, Noraziah Ahmad and Roslina Sidek, 2010. Tree based test case generation and cost calculation strategy for uniform parametric pairwise testing. J. Comput. Sci., 6: 542-547.

Mao, Y., 2010. A semantic-based genetic algorithm for sub-ontology evolution. Inform. Technol. J., 9: 609-620.

Alshraideh, M., 2008. A complete automation of unit testing for javascript programs. J. Comput. Sci., 4: 1012-1019.

Nazif, H. and L.S. Lee, 2010. Optimized crossover genetic algorithm for vehicle routing problem with time windows. Asian. J. Applied Sci., 7: 95-101. DOI: 10.3844/ajassp.2010.95.101.

Razali, R. and P. Garratt, 2010. Usability requirements of formal verification tools: A survey. J. Comput. Sci. 6: 1189-1198. DOI: 10.3844/jcssp.2010.1189.1198

Yedjour, D., H. Yedjour and A. Benyettou, 2011. Combining quine mc-cluskey and genetic algorithms for extracting rules from trained neural networks. Asian J. Applied Sci., 4: 72-80. DOI: 10.3923/ajaps.2011.72.80