Meta Heuristic Search Technique for Dynamic Test Case Generation

M. S. Geetha Devasena
Assistant Professor,
Dept. of CSE
Sri Ramakrishna Engg. College

M. L. Valarmathi
Associate Professor
Dept. of CSE
Govt. College of Technology

ABSTRACT
Software testing is an inevitable activity of software development which is crucial to the software quality and consumes approximately 50% of the software development cost. Test case design is the most important activity in testing which determines software quality. The program with the moderate complexity cannot be tested completely but verified only for input situations selected as test data. Innovative methods are emerging to perform testing as a whole and unit testing in particular with minimum effort and time. Unit testing is mostly done by developers under a lot of schedule pressure since the software companies find a compromise among functionality, time to market and quality. Thus there is a need for reducing unit testing time by optimizing and automating the process. Test suite generation is an error-prone, tedious and time consuming part of unit testing. A novel technique is proposed to automatically generate test cases from the input domain using meta heuristic search technique scatter search for branch coverage criteria with respect to cyclomatic complexity measure.

Keywords
Software testing, Unit testing, Branch Coverage Criteria and Scatter Search

1. INTRODUCTION
Testing can show the existence of errors but not the non-existence of errors. The main challenges in testing are exhaustive testing is not possible, when to stop testing cannot be assessed and there is no way to show the absence of errors. With the increased pace of production schedules, the tremendous proliferation of software design methodologies and programming languages, and the increased size of software applications, software testing has evolved from a routine quality assurance activity into a sizable and complex challenge in terms of manageability and effectiveness. The major challenges to software testing in today‘s business environment are,

• Efficiency. Is the test cycle too long? How can you ensure every test is a good investment of time and money?

• Thoroughness. How can you tell when you are done testing? How can you be reasonably sure the program is bug-free?

• Resource Management. Are testing resources strategically allocated, focusing on the highest-risk elements of the software? Are the functionally central parts of the program receiving an acceptable level of testing?

In practice, unit level testing ranges from the ad hoc tests done by programmers as they are writing code to systematic white box testing, where Unit level testing is part of a every unit must be tested and documented by a QA and Test group. In either case, the tester begins with the goal of coverage, for it is the very purpose of unit level testing [1] to achieve the highest level of coverage possible. Unit testing is important because it is performed early in the development process and it is more cost-effective at locating errors. The greatest challenge of unit level testing is to identify a minimum set of unit level tests to run. In an ideal world, every possible path of a program would be tested, accounting for all executable decisions in all possible combinations. But this is impossible when one considers the enormous number of potential paths embedded in any given program. With enormous amount of possible input situations complete test is not feasible in practice.

An essential part of testing is the selection of the most error sensitive test data. A good set of test cases is one that has a high chance of uncovering previously unknown errors. A successful test run is one that discovers these errors. To uncover all possible errors in a program, exhaustive testing is required to exercise all possible input and logical execution paths. But it is neither possible nor economically feasible except for very trivial programs. Therefore, a practical goal for software testing is to maximize the probability of finding errors using a finite number of test cases, performed in minimum time with minimum effort. Because of the central importance of test case design for testing, a large number of testing methods developed over the last decades, designed to help the tester with the selection of appropriate test data. Existing test case design methods can be categorized into black-box testing and white-box testing. Black-box test cases are determined from the specification of the program under test and white-box test cases are derived from the internal structure of the software. But it is difficult to achieve complete automation of the test case design [4,9] in both the cases. Black-box tests are automated only if a formal specification exists. The tools supporting white-box tests are limited to program code instrumentation and coverage measurement due to the limits of symbolic execution. The test case design has to be performed manually and the quality of test is reliant on the tester. Thus the, manual test case design is time-intensive and error prone when done manually.

Evolutionary testing is a promising approach for automation of structural test case generation. The aim of evolutionary testing is to increase the quality of tests and achieve cost savings in software development by means of high degree of automation.

2. EXISITING SYSTEM
2.1 Random Test Data Generation
Random test data generation techniques [2] select inputs randomly until useful inputs are found. This technique may fail to find test data to satisfy the requirements because

information about the test requirements is not incorporated. The various disadvantages of this method are such as it is appropriate only for simple and small programs, many sets of values may lead to the same observable behavior and are thus redundant and the probability of selecting particular inputs that cause buggy behavior may be astronomically small.

2.2 Static Method

Static method generates test cases without execution from several constraints based on the input variables of the program under test. These methods have several drawbacks such as treatment of loops, resolution of computed storage locations and computational cost.

2.3 Dynamic Method

Dynamic test-data generation technique carry out a direct search of tests through the execution of the instrumented version of the program under test and determines test cases that come closest to satisfying the requirement. Then, test inputs are incrementally modified until one of them satisfies the requirement. Most dynamic techniques use search based software techniques.

2.4 Search based software testing

Search-Based Software Engineering (SBSE) is the application of optimization techniques (OT) in solving software engineering problems. Optimization is the process of attempting to find the best possible solution amongst all those available. The percentage of application of search based techniques to software testing is 70% as shown in Figure 1.

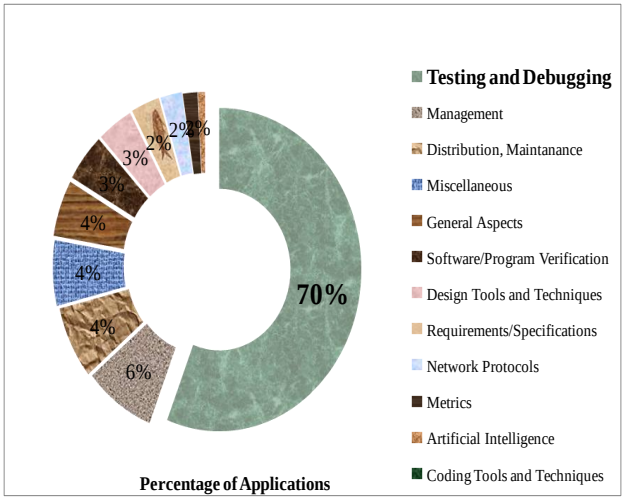

Fig 1: Application of SBSE

Software testing is a suitable candidate for Search-Based Software Engineering because the generation of software tests is an undecidable problem [14, 15] and a program's input space is very large, exhaustive enumeration is infeasible. To perform evolutionary testing, the task of test case design is transformed into an optimization problem that, in turn, is solved with meta-heuristic search techniques, such as evolutionary algorithms or simulated annealing. The input domain of the system under test represents the search space from which the test data fulfilling the test objectives under consideration is sought. The main aim of evolutionary testing is to increase the quality of the tests in addition to achieve substantial cost savings in system development by means of a high degree of automation. In various case studies, it has been proved that evolutionary testing has the potential to improve the effectiveness and efficiency of the testing process significantly. An overview of different applications of evolutionary testing is provided by McMinn [12].

2.5 Symbolic test case generation technique

Symbolic test data generation techniques are those [7, 8] that assign symbolic values to the variables to create algebraic expressions for the constraints in the program. Then constraints solver is used to find a solution for these expressions that satisfies a test requirement. The floating point inputs cannot be found by this technique because the constraint solvers cannot produce floating point constraints.

3. STRUCTURAL TESTING

3.1 Bug Statistics

The bug statistics[17] through SDLC collected from various sources given by Boris Beizer for a program of 1,00,000 lines of code shown in Table 1, among the other bugs structural bugs are the highest and half of the structural bugs are control flow and sequence bugs as shown in Figure 2.

Table 1. Bug Statistics

Size of source code: 6870000 statements		
Total Reported Bugs: 16209		
Bug Categorization	Total number of bugs	% of bugs among the total bugs
Requirements	1317	8.1
Features and Functionality	2624	16.2
Structural Bugs	**4082**	**25.2**
Data	3638	22.4
Implementation and Coding	1601	9.9
Integration	1455	9.0
System, Software and Architecture	282	1.7
Test Definition and Execution	447	2.8
Other, Unspecified	763	4.7

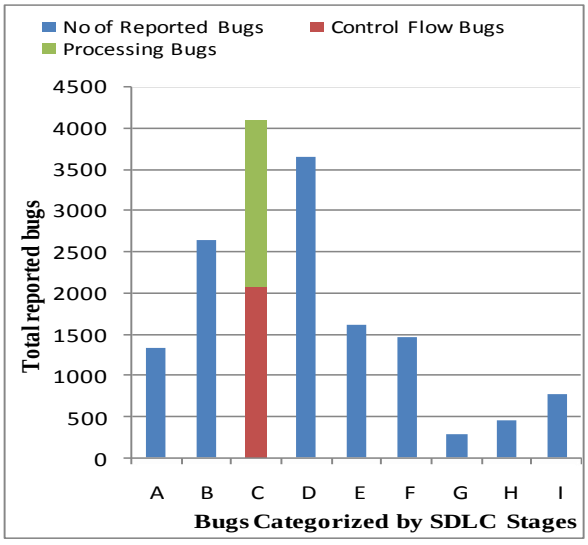

Fig 2: Bar Graph representation of Bug Statistics

The horizontal axis details of Figure 3 is mentioned below

A-Requirements
B-Features and Functionality
C-Structural Bugs
D-Data
E-Implementation and Coding
F-Integration
G-System and software Architecture
H-Test Definition and Execution
I-Other, unspecified

3.2 Cyclomatic complexity measure

Cyclomatic complexity [11, 16] (or conditional complexity) is software structural metric (measurement) used to measure the complexity of a program using Control flow graph of the program. The cyclomatic complexity of a structured program is defined as M=E-N+2P where, M- Cyclomatic Complexity, E- the number of edges of the graph, N- The number of nodes of the graph and P- The number of disconnected components. It provides lower bound on the number of test cases required to achieve branch coverage. The amount of test effort is better judged Cyclomatic Complexity. If there are fewer test cases than the measure then missing cases are to be found and more test cases than the measure shows that the coverage can be achieved with less number of test cases.

3.3 Evolutionary Testing

Evolutionary testing is characterized by the use of metaheuristic search techniques for test case generation. The test aim is transformed into an optimization problem. The search space is the input domain of the test object . The search algorithm explores the search space to find test data that fulfils the respective test aim. The neighborhood search methods such as hill climbing are not suitable in such cases. So meta-heuristic search methods are employed, e.g. evolutionary algorithms, simulated annealing, or scatter search [5, 6, 13]. In this work, evolutionary algorithms are used to generate test data because their robustness and suitability for the solution of different test tasks has already been proven in previous work [10]. Most of the previous works in applying search techniques are not taking into account float values for input domain. The first work in applying scatter search to test case generation is given by Diaz and the cyclomatic complexity is not considered [3]. The proposed work extends the previous work and applies scatter search technique to test case generation in compliance with cyclomatic complexity measure for unit testing and compares the performance with random test case generation based on the measures of test suite size and branch coverage.

4. PROPOSED SYSTEM

The proposed system develops a tool for test suite generation which takes control flow graph as input and automatically generates test cases from the input domain of various variables using scatter search technique. The architecture of the proposed work is shown in Figure 3. The Control Flow Graph Generator takes the source code of programs for which test case is to be generated and generates Control Flow Graphs.

4.1 Methodology

The various steps in the automated framework of test case generation are,

1. Taking source code under test as input CFG generator generates CFG.
2. Find the Cyclomatic Complexity measure.
3. The CFG is analyzed and the branching condition information is extracted.
4. The test cases are generated for each condition from input domain of the variables involved in the condition using scatter search technique.
5. Find the compliance of number of test cases with Cyclomatic Complexity measure.
6. The generated test cases are applied to the instrumented source code to check the branch coverage.
7. The best test cases form an effective test suite for the given source code under test.

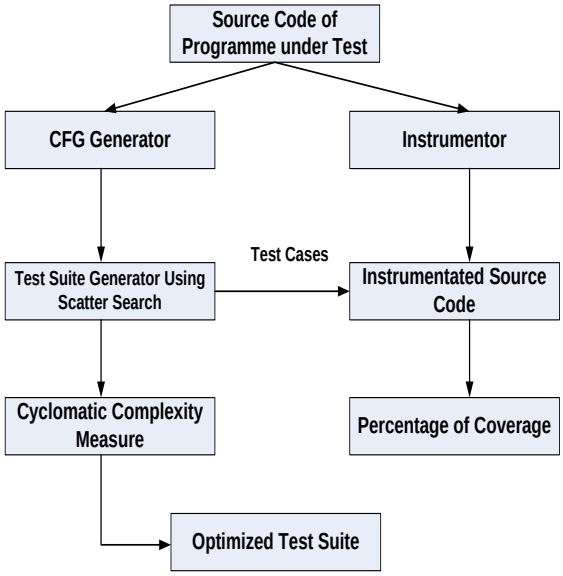

Fig 3: Flow diagram of Proposed System

4.2 Scatter search technique

The scatter search technique is a meta heuristic technique which is proven successful in real world applications such as travelling salesman problem. Recently it is found suitable for test case generation problems in software testing. But only few results have been published with relatively few samples and it must be further proven with all data types of input domain and with more samples. The scatter search algorithm is given as,

begin
Initialize Current Solution
Store Current Solution in CFG
Add Current Solution to memory list
do
Select a subgoal node to be covered
Calculate neighbourhood candidates
for each candidate **do**
calculate branch covered by candidate
endfor
if (subgoal node covered) then Add Current Solution to memory list
else Add Current Solution to memory list
endif
while (NOT all nodes covered AND number of iterations<MAXIT)
end

5. RESULTS

The proposed technique has been tested with 12 benchmarking samples including the triangle classifier program which is widely used in various research papers [1, 3, 13] in the test suite generation. The results obtained are encouraging and scatter search technique performs better than random technique. The Performance measures such as the Test Suite Size, Percentage of branch coverage are considered for comparison of the techniques. Also the test suite size is compared with the cyclomatic complexity of the program structure under test which gives the measure of test cases required to cover the program.

$$\%\ \text{of Test suite size} = \frac{\text{Total number of Test cases}}{\text{Total number of branches}} \times 100$$

$$\%\ \text{of branch coverage} = \frac{\text{100- Total no. of unfeasible branches}}{\text{Total number of branches}} \times 100$$

The results got by random technique can be given in Table 2.

Table 2. Results of Random Technique

Samples	Test Suite Size	% of Branch Coverage	Cyclomatic Complexity
S1	8	75	3
S2	5	80	2
S3	7	100	3
S4	3	100	2
S5	9	77.77	3
S6	11	81.8	3
S7	5	100	2
S8	6	100	3
S9	5	100	2
S10	8	87.5	3
S11	10	88.88	3
S12	15	93.33	4

The results show that the branch coverage varies from75% to a maximum of 100% and that is achieved with more number of test cases than the calculated Cyclomatic Complexity measure. The results got by scatter search technique are given in Table 3.

Table 3. Results of Scatter search Technique

Samples	Test Suite Size	% of Branch Coverage	Cyclomatic Complexity
S1	3	100	3
S2	2	100	2
S3	3	100	3
S4	2	100	2
S5	3	100	3
S6	3	100	3
S7	2	100	2
S8	3	100	3
S9	2	100	2
S10	3	100	3
S11	2	88.88	3
S12	3	93.33	4

It is found that branch coverage is increased by 10 percentage and test suite size is reduced by 67 percentage. It is achieved with as many numbers of test cases as calculated by Cyclomatic Complexity measure. The performance analysis graph based on the number of test cases in the test suite and the percentage of branch coverage of both the techniques is given in Figure 4 and Figure 5 respectively.

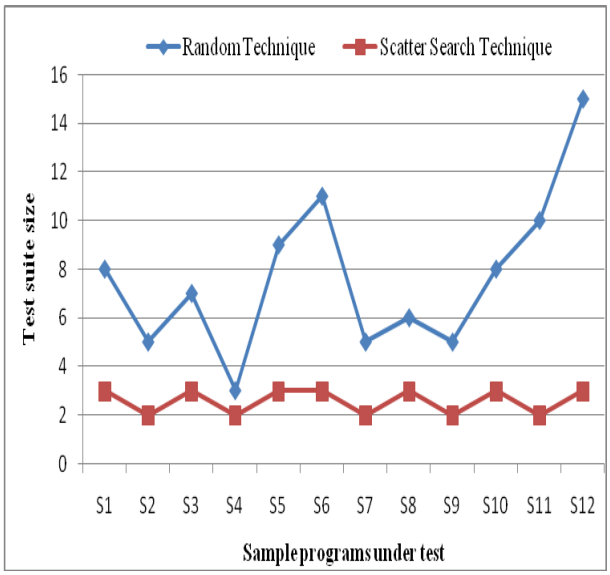

Fig 4: Test Suite Size Comparison

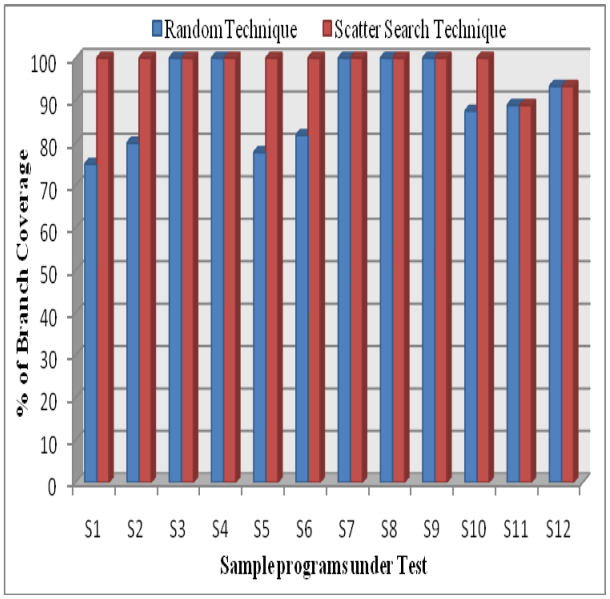

Fig 5: Percentage of Branch Coverage Comparison

6. CONCLUSION

Software testing is an important activity and critical too in deciding quality of the software. Test suite generation is vital part of testing process which determines the quality of test. This technique of automated generation of test cases from the input domain can assist the developers and testers with error-sensitive test data and helps to perform unit testing with minimum time and resources. Also the optimized number of test cases generated is much helpful in regression testing which otherwise carried out with greater number of test cases. The technique can be further extended for multiple coverage criteria. Also the effectiveness can be further proven with fault detection effectiveness.

7. REFERENCES

[1] Chilenski1, John Joseph Chilenski and Steven P. Miller, 1994. 'Applicability of Modified Condition/Decision Coverage to Software Testing', Software Engineering Journal Vol. 9, No. 5, pp.193-200.

[2] Edvardson, J. 1999. 'A Survey on Automatic Test data generation', In proceedings of the second conference on computer science and engineering Vol.2, No.1, pp.343-351.

[3] Eugenia Diaz, Javier Tuya, Raquel Blanco, Jose Javier Dolado, 2008. 'A tabu search algorithm for structural software testing', Computers and Operations Research Vol. 14, No. 3, pp.38-69.

[4] Ferguson and Korel, B. 1966. 'The chaining approach for software test data generation', ACMTOSEM vol. 5, pp.63-86.

[5] Glover, F. 1989. 'Tabu search: part I', ORSA Journal on Computing, Vol. 3, No.1,pp.190-206.

[6] Glover, F. 1990. 'Tabu search: part II', ORSA Journal on Computing, Vol. 4, No. 2, pp.4–32.

[7] Howden, W.E. 1977. 'Symbolic testing and the DISSECT symbolic evaluation system', IEEE Transactions on Software Engineering vol.3, no. 4, pp. 266-278.

[8] John Clarke, Mark Harman, Bryan Jones. 2000. 'The Application of Metaheuristic Search techniques to Problems in Software Engineering', IEEE Computer Society Press Vol.42, No.1, pp.247-254.

[9] Lindquist, T.E. and Jenkins, J.R. 1998.'Test-case generation with IOGen', IEEE Software vol.5, no.1,pp. 72-79.

[10] Lin, Yeh, P.L. 2001. 'Automatic test data generation for path testing using Gas', Information Sciences Vol. 4, No.13, pp. 47-64.

[11] McCabe, Tom, 1976. 'A Software Complexity Measure', IEEE Trans. Software Eng Vol.2, No.6, pp.308-320.

[12] McMinn, p. 2004. 'Search Based Software Test Data Generation:A survey', Journal on Software Testing, Verification, and Reliability vol.14, no.2, pp.105-156.

[13] Raquel Blanco , Javier Tuya , Belarmino Adenso-Díaz. 2009. 'Automated test data generation using a scatter search approach', Information and Software Technology Vol. 51, No.1, pp. 708-720.

[14] Tao Feng, Kasturi Bidarkar. 2008. 'A Survey of Software Testing Methodology' vol.25, no-3, pp.216-226.

[15] Voas, J.M, Morell, J. and Miller, K.W. 1991. 'Predicting where faults can hide from testing', IEEE vol: 8, pp, 41-48.

[16] Wegener, Baresel DeMillo RA, Offutt, A.J. 1991. 'Constraint-based automatic test data generation' IEEE Transactions on Software Engineering Vol.17.

[17] Boris Beizer. 2000. 'Software Testing Techniques', Second edition.