Software Coverage Analysis using Hybrid Approach

R.J.Sarath Prabath[1], M.Siddardha[2], M.Bapi Raju[3], G.Rajesh[4] and Praveen R Srivastava[5]
Birla Institute of Technology and Science-Pilani, Pilani.

I. ABSTRACT

Software Testing is one of the most critical phases in development of software. The aim of software testing is to create quality software products to meet the expectations of an organization. One of the problems encountered during Software testing phase of software development is determining when the testing is to be ended and the actual software can be delivered to the client. The required criteria to be achieved, for the tester to gain confidence in software can be attributed to the term "Software Coverage".

The proposed methodology provides an analysis of path coverage. The proposed algorithm uses Genetic Algorithms and the technique has been optimized with the aim to gain maximum test coverage and minimize the complexities involved in testing.

***Key Words*:**
Software coverage, mutation, cross-over, control-flow-graph, population, non-dominant sorting, fitness function, generation, fronts.

II. INTRODUCTION

Topic Background:

Software Testing is the main component of Software Engineering.[1] Approximately 50% of the time and cost are spent for testing purpose only[3]. Bad testing methodologies may delay the whole process of software development and also gives a low Quality Software as output. Hence appropriate Testing methodology is to be chosen so as to reduce the effort and cost.

According to a study [1], software manufacturers in United States of America mislay around a total of 21.2 billion dollars because of non-optimized testing. Ever growing competition in the software industry demands software testing process to be cost effective and efficient.

Testing all the valid test cases for a particular software definitely takes a lot of time [3]. Therefore optimizing the number of test cases before actual testing process begins is essential. This is where concept of **Software Coverage/Code Coverage** comes into picture. Code coverage is a measure used in software testing which describes the degree to which the source code of a program has been tested.

A better Code coverage increases confidence of developer in quality of software. And a decision to discard the testing phase is based on software coverage only [1].

As mentioned above, the coverage has to be maximized for better testing method. It can be done by using any one of the following optimization techniques alongside genetic algorithm[2]:

- Single Objective
- Multi-Objective

Although a Single objective optimization technique can be used, algorithm uses a two-objective optimization technique. It is mainly because of the reason that though a long test case can ensure high coverage, it is not preferred as it also increases the complexities involved in testing.
Therefore, a variant of NSGA-II, a multi-objective unconstrained optimization algorithm for division of test cases into fronts is used (dominant test cases which yield better coverage).
While a normal Genetic Algorithm gives a large number of test cases at the end of certain number of generations, it is impossible to deal with all of them where as use of NSGA-II can limit the number of test cases at the end.[2]

III. PROPOSED MODEL:

A source-code triggered software is taken as the input for measuring the coverage. Each code block is given a block number and then modelled as Control Flow Graph (CFG) to derive the basic set of test paths for testing.
The number of test paths varies for different software. Our proposed model takes an initial set of valid test cases, let us suppose from some other testing team.(Generation 0) and sent for Coverage analysis. If the calculated coverage meets the specified coverage, then Generation 0 can be called Optimal and further computations can be called off. Else, algorithm proceeds to reproduce the next generation of test cases and analyzing the formed test cases again until the required coverage is achieved.
As our model uses a two objective optimization technique, two objective functions for each test case are to be defined.

The first function being the coverage function of test case, and the second takes the complexities involved in each block of software, for the calculation.
objective function1– is the percentage of test paths covered by a test case. As the respective test paths for a given software has been declared beforehand, F1 determines the number of test paths a test case covers and returns the percentage of test paths covered by that particular test case

$$F1(X) = \#test\ paths\ covered/total\ number$$

objective function2– it's the sum of the complexities involved in testing the software using that particular test case. Every source code triggered block is assigned a numerical value depending on the complexities involved in testing that particular code block. The three complexities taken for our project are time Involved in testing, cost incurred for the testing. Weightage for the above factors based on the requirements of the software will be assigned.

$$F2(X) = \sum (weightage * Numerical\ value\ assigned\ to\ block)$$

after F1 and F2 values for each test case in a generation is calculated independently, move on to the next step of the project i.e., application of Non-dominant Sorting

algorithm to divide the test cases in a generation to a number of fronts.
Our objective is to maximize F1 () and minimize F2 ().

Non-Dominant Sorting:
Test cases, whose F1 and F2 values are determined are fed into this module. These test cases are then compared based on their objective function values and divided into different fronts, based on this algorithm.
Our objective is to *maximize F1() and minimize F2().* The divided fronts are in the order of their importance ie.., first front contains the best test cases available and quality degrades when going down to the next front.

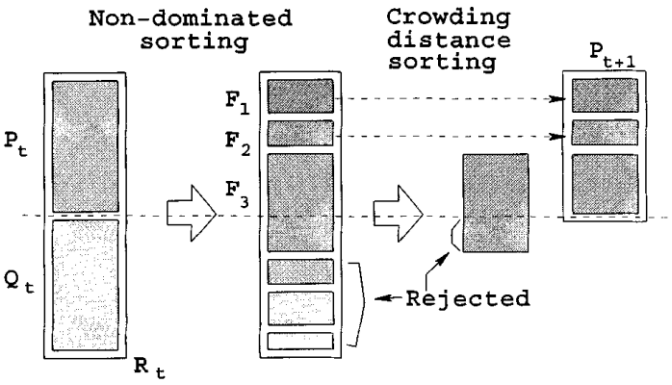

Fig 3.1Non-Dominant Sorting

As shown in above figure 3.1 if (P+Q) number of test cases are passed to this module, this module compares all the test cases and divides the input cases into various fronts.
Therefore, if one needs to select only a part of the input test cases based on their objective function values, the top few fronts can be taken and can discard the remaining. But in case, where only a certain portion of a front is to be selected, Crowding Distance is used, which is explained below. The figure 3.1 depicts the same.

Crowding Distance:
The test cases in the same front after Non-Dominant Sorting can be called equally important. So, to choose between test cases of a same front, this function is to be called.
From the corresponding front, a test case with least coverage is chosen and distance between that point to every other point is calculated. That distance is sorted and till the required numbers of test cases are reached, take the top test cases from the sorted list and return them as output.
This function is not meant for the betterment of coverage but is meant to bring variation among the current generation test cases, so that next generation is better off. Thus, it ensures the variation of test cases.

From figure 3.1, it can be seen that for given P+Q number of test cases, top P test cases are obtained and the remaining Q number of test cases are discarded. The newly obtained P numbers of test cases are taken as current generation and reproduction operators are applied to produce next P+Q' set of test cases.

Cross-over and Mutation:
The test cases in the current generation are taken to produce the next generation using cross-over and mutation.
In Cross-Over method, two test cases are taken from current population randomly and at another random position they are cut and appended to produce a new test case.
In the case of mutation, one or two or three blocks in a test case are replaced with some random numbers, to get a new test case.
The new test cases generated are checked for their validity and if they are valid, are appended to the population of next generation. Therefore, the population of next

generation contains the population of current generation and newly formed valid test cases.

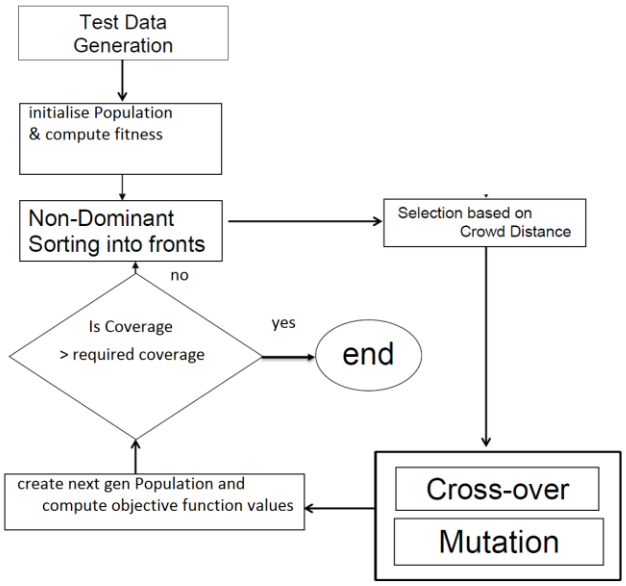

Fig 3.2 Flow graph of proposed algorithm

In *Fig 3.2*, it can be seen that the algorithm takes the set of valid test cases as initial population. Algorithm calculates the F1 and F2 values (fitness functions) and checks for the terminating condition. If it fails, *Cross-over and Mutation* function is called to produce the next intermediate generation.
Each test case produced after reproduction are checked for their validity also. For each test case in next intermediate generation fitness functions are calculated and then *Non-Dominant Sorting* and *Crowding Distance* functions are called to divide the population to fronts and select the top test cases as the next generation population and the whole loop is run again.

IV. ANALYSIS

In this section, an example analyzing the coverage for a particular software is provided. As you can see below, the Control Flow Graph for that particular software is shown. In addition the results b/w traditional Genetic Algorithm are compared with our proposed Algorithm

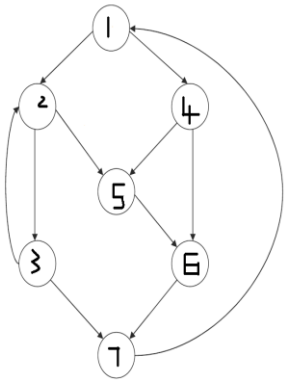

Figure 4.1 Control-Flow Graph for a software

Path 1(p1): 1,2,3,7.
Path 2(p2): 1,2,3,2,3,7
Path 3(p3): 1,2,5,6,7.
Path 4(p4): 1,4,5,6,7.
Path 5(p5): 1,4,6,7.
This now forms the basis set of paths for the graph in Figure 1.7. In theory, if we allow for the basic notions of scalar multiplication and addition, one should now be able to construct any path from our basis. Let us attempt to create a 6^{th} path: 1,2,3,2,5,6,7. This is the basis sum p2 + p3 – p1. This equation means to concatenate paths 2 and 3 together to form the path 1,2, 3, 2, 3, 7, 1, 2, 5, 6,7 and then remove the four nodes that appear in path 1, resulting in the required path 6.

Coverage analysis for this particular software is done by NSGA-II as follows:

After Generation 1:

Population:
[[1, 5, 3, 2, 5, 6, 5], [1, 2, 3, 2, 3, 7, 1, 4, 6, 7], [1, 4, 5, 6, 7, 1, 4, 6, 7], [1, 4, 5, 6, 7, 1, 2, 3, 7], [1, 4, 5, 6, 7, 1, 2, 3, 2, 3, 7]]

So, as per the algorithm, the above population is sorted into fronts as shown below to select the dominant ones in order to fetch the maximum possible coverage.

Fronts:
Front 0: [1, 5, 3, 2, 5, 6, 5]
[1, 2, 3, 2, 3, 7, 1, 4, 6, 7]
[1, 4, 5, 6, 7, 1, 4, 6, 7]
[1, 4, 5, 6, 7, 1, 2, 3, 7]
[1, 4, 5, 6, 7, 1, 2, 3, 2, 3, 7]

Average Coverage is calculated and found to be: 56.0 , which is quite low.

After Generation 2:
Population:
[[1, 5, 3, 2, 5, 6, 5], [1, 2, 3, 2, 3, 7, 1, 4, 6, 7], [1, 4, 5, 6, 7, 1, 4, 6, 7], [1, 4, 5, 6, 7, 1, 2, 3, 7], [1, 4, 5, 6, 7, 1, 2, 3, 2, 3, 7], [1, 2, 3, 2, 3, 7, 1, 4, 5], [1, 4, 5, 6, 7, 1, 3, 2, 3, 7], [1, 4, 5, 6, 7, 1, 2, 3, 2, 5, 6, 5]]

Fronts:
Front 0: [1, 5, 3, 2, 5, 6, 5]
[1, 2, 3, 2, 3, 7, 1, 4, 6, 7]
[1, 4, 5, 6, 7, 1, 4, 6, 7]
[1, 4, 5, 6, 7, 1, 2, 3, 7]
[1, 4, 5, 6, 7, 1, 2, 3, 2, 3, 7]
[1, 4, 5, 6, 7, 1, 3, 2, 3, 7]
Front 1:
[1, 4, 5, 6, 7, 1, 2, 3, 2, 5, 6, 5]
[1, 2, 3, 2, 3, 7, 1, 4, 5]

Average Coverage: 52.5

After producing considerable no: of generations, the required software coverage value is reached. In this example it was achieved at Generation 9.

Achieved Coverage = **86.67%**.

So, to view the alteration of coverage values after every generation, let us plot the graph showing the population and it's corresponding fetched coverage.

X-axis – Generation number of population
Y-axis – Average Coverage percentage for every generation

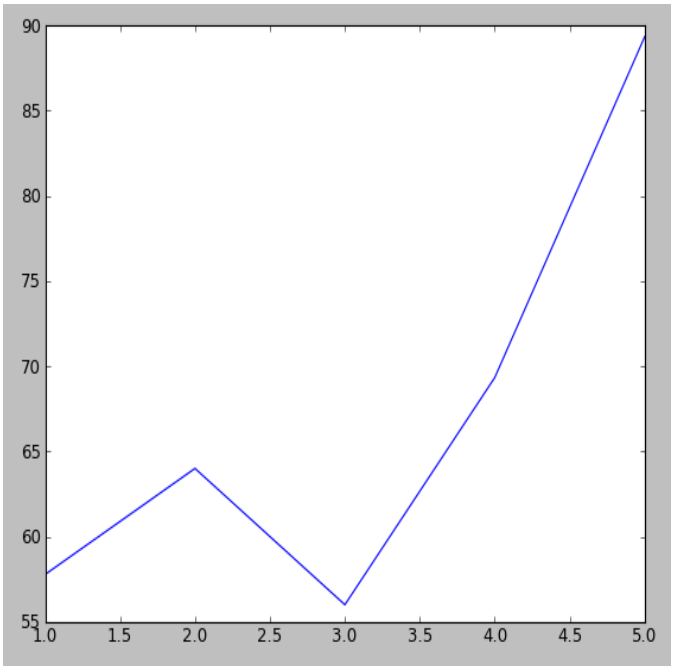

Fig 1.1 Graph showing the coverage values achieved after every generation using NSGA-II

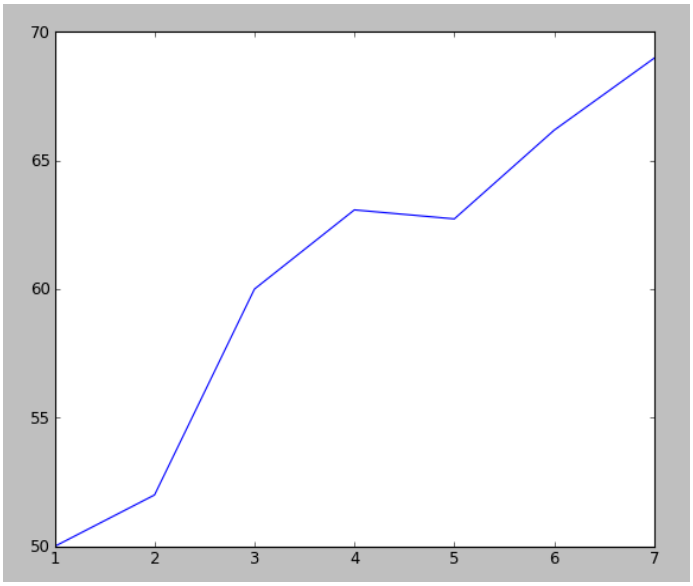

Fig 1.2 Graph showing the coverage values after every generation using traditional Genetic Algorithm

So, one can observe that even after seven generations in traditional Genetic Algorithm approach we get a maximum of 68%. Contrary to this, using NSGA-II a coverage of 86.67% was achieved just after five generations which is highly efficient.

Conclusion:

The proposed algorithm is a computationally fast algorithm to obtain the maximum possible software coverage for a software. The proposed algorithm is much superior to traditional Genetic Algorithm as it limits the number of test cases generated at the end of certain number of generations.
The number of test cases is reduced drastically compared to traditional Genetic Algorithm.

References:

[1] *Coverage* Analysis for GUI Testing *By Abdul Rauf, MS (CS), MSc (CS)*
Department of Computer Science, National University of Computer & Emerging Sciences, Islamabad, Pakistan july-2010

[2] A Fast and Elitist Multiobjective Genetic Algorithm:NSGA-II
Kalyanmoy Deb, Associate Member, IEEE, Amrit Pratap, Sameer Agarwal, and T. Meyarivan
182 IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION, VOL. 6, NO. 2, APRIL 2002

[3] Application of Genetic Algorithm in Software Testing
Praveen Ranjan Srivastava1 and Tai-hoon Kim2
International Journal of Software Engineering and Its Applications
Vol. 3, No.4, October 2009

[4] Path Testing
Luke Gregory ,Professor H. Schligloff and Dr. M. Roggenbach

[5] A Review of ZSelection Methods in Genetic Algorithms
Dr.T.Ravichandran
Hindusthan Institute of Technology, Coimbatore-641032, Tamil Nadu, India.
R.Sivaraj et al. / International Journal of Engineering Science and Technology (IJEST)

[6] Baxter, Branch Coverage for arbitrary Languages made easy

[7] Xie, Improving Automation in Developer Testing: State of practice ,North Carolina State University Feb 20, 2009.

[8] A Fast Elitist Multiobjective Genetic Algorithm:NSGA-II
ARAVIND SESHADRI