Global Journal of Computer Science and Technology: C
Software & Data Engineering
Volume 14 Issue 8 Version 1.0 Year 2014
Type: Double Blind Peer Reviewed International Research Journal
Publisher: Global Journals Inc. (USA)
Online ISSN: 0975-4172 & Print ISSN: 0975-4350

An Overview of Recent Trends in Software Testing

By Anupama Surendran & Philip Samuel

Cochin University of Science & Technology India

Abstract- In the field of search based software testing, genetic algorithm based testing has received a major share of attention among researchers during the last few years. Though there are advantages for this type of testing, there also exist some practical difficulties which can make this technique less attractive for software testing industry. The potential of program slicing in testing has not been fully exploited till now and the works that have explicitly demonstrated the application of slicing in testing field are rare. Our paper aims to analyze existing techniques for software testing and to introduce an approach for software testing using program slicing technique. A systematic review of genetic algorithm based works reveals that, fitness function design, population initialization and parameter settings impact the quality of solution obtained in software testing using genetic algorithm. Based on the conclusions from the existing literature, we have probed deeper about the issues in these areas. Making an unbiased review like this may help to solve these unresolved issues in genetic algorithm based software testing. In this work, we have emphasized and has given clear directions on how slicing can be used as a potential tool for practical software testing. In addition, a set of research questions have been framed, which may be answered by reviewing the study made in this work. This may help future research in this area, leading to major breakthrough in software testing field.

Keywords: program slicing, software testing, forward slicing, genetic algorithms.

GJCST-C Classification: D.2.5

ANOVERVIEWOFRECENTTRENDSINSOFTWARETESTING

Strictly as per the compliance and regulations of:

© 2014. Anupama Surendran & Philip Samuel. This is a research/review paper, distributed under the terms of the Creative Commons Attribution-Noncommercial 3.0 Unported License http://creativecommons.org/licenses/by-nc/3.0/), permitting all non-commercial use, distribution, and reproduction inany medium, provided the original work is properly cited.

An Overview of Recent Trends in Software Testing

Anupama Surendran [α] & Philip Samuel [σ]

Abstract- In the field of search based software testing, genetic algorithm based testing has received a major share of attention among researchers during the last few years. Though there are advantages for this type of testing, there also exist some practical difficulties which can make this technique less attractive for software testing industry. The potential of program slicing in testing has not been fully exploited till now and the works that have explicitly demonstrated the application of slicing in testing field are rare. Our paper aims to analyze existing techniques for software testing and to introduce an approach for software testing using program slicing technique. A systematic review of genetic algorithm based works reveals that, fitness function design, population initialization and parameter settings impact the quality of solution obtained in software testing using genetic algorithm. Based on the conclusions from the existing literature, we have probed deeper about the issues in these areas. Making an unbiased review like this may help to solve these unresolved issues in genetic algorithm based software testing. In this work, we have emphasized and has given clear directions on how slicing can be used as a potential tool for practical software testing. In addition, a set of research questions have been framed, which may be answered by reviewing the study made in this work. This may help future research in this area, leading to major breakthrough in software testing field.

Keywords: program slicing, software testing, forward slicing, genetic algorithms.

I. Introduction

In God we trust, everything else we test . This famous quote conveys the idea that almost all the things in this world are unreliable without testing [6]. Proper testing makes the software robust and trustworthy and hence the importance of testing cannot be overemphasized. From simple home appliances and common automobiles, to life support devices like mechanical ventilators and mission critical systems like nuclear reactors, there is an unending list of components which depend on some form of software for their proper functioning [27]. These softwares in turn depend on testing for their infallibility. Imagine a pharmaceutical company introducing a new drug in the market without proper trials and testing. It is not only illegal, but also extremely unsafe and potentially deleterious. Similarly, software development without testing makes it unreliable, unusable and even unsafe.

While one of the main intentions of software testing is to check for and identify errors in software, a software tester has a much wider gamut of responsibilities. For example in our real life activity, in an automobile where there is a sound due to the loosening of wheel, the defect may be corrected by tightening it, but the alignment of the tightened wheel may not be synchronous with the other wheels. Therefore in the next step, the wheels are to be aligned for the proper running of the vehicle. Similarly, finding the root cause or in other words, finding the dependency during software testing is one of most challenging aspects of software testing as rectifying an error may introduce some side effects in the software. Getting the dependency relations present in a program serves as the backbone of several other processes in software development, such as regression testing, program comprehension, maintenance, reverse engineering and re-engineering [16, 17]. This implies that, though software testing can be very challenging, it has a very significant influence and marked relevance in software development industry. In the earlier days, most of the applications used simple software and they were mostly standalone applications. The nature of modern day software can make its testing not an easy task. Many of the software used nowadays is real-time and embedded software with web interface. This type of software may have several interconnected modules and such software needs to be continuously tested until they get outdated from the world market. Technological changes, requirement changes and platform changes raise the need for continuing testing in such systems. In such software, the software dependency consideration is an unavoidable factor which decides the reliability of the software. Even a minor error may cause great mishap in such software applications. The unrestricted size of the source code is another problem plaguing the software testing industry. In the case of large commercial software, there will be several modules and lines of code which make software testing process more difficult. As testing cost increases with source code size, it should be one of the primary concerns of the software tester. In the field of software testing, a software tester cannot leave the scene after finishing the testing process [31]. During software testing, the test cases designed for solving the error in some part of the source code may prove to be insufficient to solve the bugs occurring some other parts of the source code. This is similar to the creation of mutant species. For example, long term use of an insecticide against a particular species of insect, makes it vulnerable to development of

Author α σ : Cochin University of Science & Technology, Kochi, Kerala, India. e-mail: anudeepaknair@gmail.com

resistance by genetic modification and mutation in the insect. In such a situation, new insecticides have to be used to kill that insect. Similarly, the test cases designed for a particular test scenario may fail in some situation. This may be due to the changes made in the source code or due to the change in design requirements made as per user specifications. New test cases are to be found for solving such problem or the existing test cases should be updated by the software tester. From the above discussion it is evident that, a good tester should be a good software designer, an intuitive code developer and a reliable maintenance person, all rolled into one. For example, consider the situation where a company decides to change its product as per user requirements. Now, the software designer and code developer can fulfill their parts just by completing the work in their respective areas of expertise. On the other hand, for the testing to be fully reliable, the tester has to understand the changes made by designer and code developer and then develop appropriate testing methods. Truly speaking, a good software tester has to be a skilled all-rounder.

Several methods were developed with an aim to address the challenges existing in software testing industry. Among the different software testing strategies, search based testing has received immense attention and especially, genetic algorithm based testing has made a marked influence in software testing research [30]. This is due to the adaptability of genetic algorithms to handle the testing process and the ability to represent the software testing problem as an optimization problem [38]. Considering the volume of work done in genetic algorithm based software testing, it is crucial to identify the merits and demerits of this approach. Even though genetic algorithm based testing has made a great impact in academic research, only very little attention has been given to understand the complexities of using genetic algorithms in practical software testing. This work focuses on this and we have tried to highlight the challenges involved in genetic algorithm based approaches for using it as a practical tool in software testing. The main reason for choosing this problem in our work is because of the usage of genetic algorithms in software testing without knowing the ambiguities in genetic algorithm based testing. In this paper, we have mentioned some works which utilize genetic algorithm for testing [38, 39, 40, 44, 51, 52, 54]. We can see that none of these works have adopted any general operator setting for testing purpose. The inherent non-deterministic nature of the genetic operators makes the program testing process a demanding task. The strength of using genetic algorithm mainly depends on setting the genetic parameters to their appropriate values and this in turn depends on the problem to be solved. This itself is a major challenge faced by testers.

In this work, we have suggested a program slicing approach for software testing and have highlighted the strengths of using program slicing as a tool in software testing industry. It was Weiser who introduced slicing in 1979 [15, 53] and his work encouraged many research works developing slicing algorithms. According to Weiser, slicing criterion consists of two parameters and it is represented as (V, n), where ‘V‘ is a set of variables and ‘n‘ is the program point [53]. In program slicing, source code size is minimized by converging focus on some specific program part implied by the ‘slicing criterion‘ [20,49]. This property of slicing is highly relevant, as source code size is a major concern is modern day software. Instead of analyzing the whole program, slicing reduces the program search space which in turn minimizes the testing effort. Setting the slicing criterion with respect to the variable with incorrect value can help to identify the portion of source code which causes error during program testing. Here the manual effort of the program tester is reduced considerably as there is no need to consider the whole source program [11, 47]. Slicing also helps to trace program dependencies which are very crucial in testing. In several works it has been mentioned that program slicing may be used for testing purpose [17, 20, 21]. None of these works gave a clear picture of how to utilize slicing to make testing more meaningful. Apart from program testing, slicing can be used in several applications such as program debugging [34, 53], program comprehension [22] and program maintenance [17]. In this paper, we have demonstrated a forward slicing approach for testing and have tried to mark the merits of program slicing based testing approaches.

Finally, this paper aims to:

- Introduce program slicing as a major research direction in software testing
- Present an analytical description of program slicing and to demonstrate how it can be applied in software testing
- Assess the current research trends in software testing with a special focus on genetic algorithm based testing
- Analyze the shortcomings and challenges for making genetic algorithm based approaches practical in software testing industry
- Highlight the significance of program dependency in software testing, and explain how program slicing can effectively resolve this issue

The remaining section of the paper is organized as follows. Section 2 gives the basics of program slicing and genetic algorithm. Section 3 compares program slicing based testing and genetic algorithm based testing approaches. Based on the observations made in section 3, some research questions are framed in section 4. In section 5, we have given an explanation of the research questions in section 4. Threats to validity of this work are given in section 6 and section 7 gives the conclusion.

II. Basics

As we are doing a detailed study of genetic algorithm based and program slicing based software testing methods, we shall go through the basic principles of genetic algorithm and program slicing concepts. Based on the conclusions from the exiting literature, we will have to probe deeper about the issues in these areas. Making an unbiased review like this may help to solve the issues in genetic algorithm based software testing and at the same time help to understand the relevance of program slicing in software testing. This may help the future researchers working in this area.

a) Genetic Algorithms

In order to conduct a proper review of genetic algorithm based software testing, it is essential that one should be familiar with the basic concepts and terms in genetic algorithm. This is dealt with in this section. Genetic algorithm is a type of evolutionary algorithm and is considered as the best and the strongest of all evolutionary algorithms [18, 24]. It is a type of search technique developed by John Holland and works on Darwin's principle of survival of the fittest. Genetic algorithm uses the technique of natural genetics, representing a computer model of biological evolution. Genetic algorithms have the ability to solve a variety of optimization and search problems. Several testing techniques use genetic algorithms believing that testing may be carried out in a better way using the natural evolutionary process present in them [39].

Genetic algorithm identifies an optimal solution for a problem by applying natural evolutionary techniques to a group of possible solutions referred to as ―population [18, 40]. After each generation, a new generation is formed which is better than the previous generation. The series of steps involved in genetic algorithm are population initialization, selection, crossover, mutation and termination. A string of digits called chromosomes are present and each individual of the string is called a gene. Each individual in the population has a fitness value which decides the quality and performance of that individual. Greater the fitness value better will be the problem solving capacity of an individual [25]. Collection of chromosomes makes up a population. The initial population is created randomly and the fitness of the individuals in the population is calculated. This information is used to select the best candidates for forming the next generation parents. After selecting parents of the successive generation, the next step is to combine these candidates to form the offspring. Crossover operation is used to perform this [36, 54]. Crossover enables the selection of good features from parents to form the offspring. Mutation is applied to the offspring to create better quality individuals. Mutation is defined as the process of altering the genes in the chromosome [43]. A new generation is chosen from the offspring based on the fitness of the individuals. These individuals are considered as parents of the next generation. This cycle is repeated until a global solution for the problem is obtained. The basic steps of genetic algorithm are given in algorithm 1.

ALGORITHM 1

```
procedure Genetic Algorithm
begin
GET (Initial Population);
CALCULATE FITNESS (Initial Population)
loop
FINALZE POPULATION FOR CROSSOVER (Parent population)
PERFORM CROSSOVER (Parent population, child)
APPLY MUTATION (Child)
CALCULATE FITNESS (Child)
GET NEXT GENERATION (Parent population, Child)
stop process when TERMINATION CRITERA
exit loop
end
```

b) Program Slicing

This section deals with some of the common terms in program slicing. Slicing is defined as the process of deleting all those statements from a program which cannot affect the values of a variable of interest. In other words, a slice is a subset of source program statements. Slicing is performed based on slicing criteria. A slicing criterion comprises a program location and a set of variables known as slice set. If P is a program, x is a statement in P and y is a variable in P, then the slicing criterion (C) is given as C= (x, y). Program slicing can be divided into various types. Based on slicing criteria, the two main types are static and dynamic slicing [32, 35], while based on direction of slicing the two main types are forward and backward slicing [22, 49].

i. *Static Slicing*

A slice constructed by ignoring those parts of the program that are not relevant to the values stored in the chosen set of variables at the chosen point is known as static slice [8, 34]. As mentioned above slicing criterion C= (x, y), where x is a statement in the P (program) and y is a variable in P. Given a variable 'y' and a point of interest 'x', slice will be constructed for y at x. An example program is given in table 1, where the static slice criterion is given as <11, a>. The result will be the set of statements <4, 5, 6, 8, 9>. Backward slicing gives all the program statements which affect the value of a particular variable at a particular point [TIP 1995]. Forward slicing gives all the program statements which are affected by declaring a variable at a given point in the program [22, 29].

Table 1 : Static slicing

Program Statements	Static slice for criterion <11, a>
1 main()	4 cin>> b;
2 {	5 a = 0;
3 int a,b;	6 while (b <= 10)
4 cin>> b;	8 a=a+b;
5 a = 0;	9 ++ b;
6 while (b <= 10)	
7 {	
8 a=a+b;	
9 ++ b;	
10}	
11 cout<< a;	
12 cout<< b;	
13 }	

ii. *Dynamic Slicing*

The concept of dynamic slicing was given by Korel [33]. The set of statements that affect the value of a variable for one specific input is known as dynamic slice. In dynamic slicing we have to consider three parameters. First one is the point of interest within the program, second one is the variable and the third one is the sequence of input values for which the program was executed. Dynamic slicing criterion is defined as C= (x, y, i). Here x is the statement in the program, y is the subset of variables in the program and i is the input value [11]. A sample program to be sliced is given below in table 2. The variable with respect to which slicing is to be done is p, slicing point is the end of the program and input given is n=0.

Table 2 : Dynamic slicing

Program Statements	Dynamic Slicing Criterion :-(10, p, n=0,)
1 scanf("%d",&n);	p=0
2 s=0;	
3 p=0;	
4 while (n>0)	
5 {	
6 s=s+n;	
7 p=p*n;	
8 n=n−1;	
9 }	
10 printf ("%d%d", p, s);	

In static slicing though the size of the slices obtained will be large, all possible executions will be considered. On the other hand, in dynamic slicing the down side of small size of slices is that the result will be focused only for a specific input [32].

III. Evaluation of Testing Approaches

This section analyses the testing approach based on genetic algorithm and introduces our approach based on program slicing. Here we have identified some points to justify our analysis and these are used to frame the research questions in section 4. We have divided this section into three parts. In the first part the purpose of software testing is explained. The second part deals with genetic algorithm based software testing. Some relevant works in that field and our observations regarding genetic algorithm based testing are given in this section. In the third part we have introduced our program slicing based testing approach and have described its benefits and importance.

a) *Software testing*

The section gives an insight into the basics of software testing. In software testing the target program is executed to identify the errors. This is followed by debugging to rectify the identified errors [21]. Before starting the testing process, the objectives or the goals should be properly set and the tester should be aware of

© 2014 Global Journals Inc. (US)

the strategy to be followed to achieve the set goals [10]. It is very essential that the tester should have an idea of user requirements and should also be able to identify the conditions which will have an adverse effect on the selected testing strategy. The main objectives of testing are [4, 41]

- To affirm that the software developed is error free
- To check whether the developed software is functioning correctly according to the program developer and program tester
- To confirm that the developed software works correctly without causing any data loss.

Therefore developing an effective method for testing is an inevitable part of all software systems

b) Genetic algorithm based testing

In the past few years, search based software testing, especially evolutionary algorithm, has gained immense popularity [2, 9]. A graph is shown in figure 1, which shows an increase in rate of publications and research works in search based software testing during the period 1975 to 2010[37]. Among evolutionary algorithms, genetic algorithm is one of the widely researched techniques for software testing. They are included in dynamic testing techniques [26]. In dynamic testing, the program is executed based on given input data to obtain the corresponding output, while in static testing, the program has to be analyzed line by line to check for the errors in the program. Thus in static testing, the ability to find errors depends on the tester's experience.

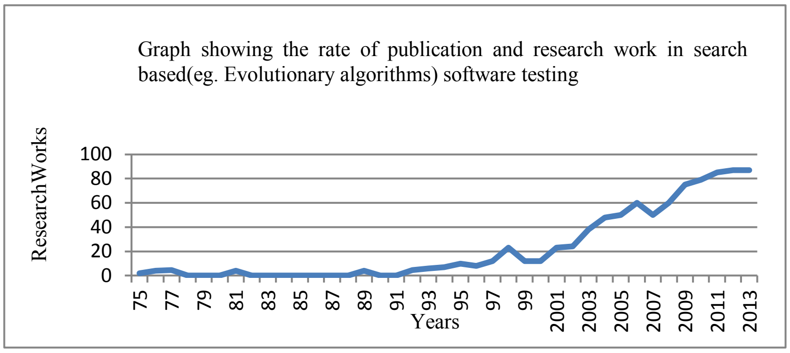

Figure 1 : Research works in search based software testing during the period 1975 to 2013

Genetic algorithms are used to perform automated software testing due to their ability to represent the testing problem as an optimization function. Finding a solution for this optimization problem gives a solution for the testing process also. There were several attempts to generate test data using single population, multiobjective, master-slave, fine-grained and coarse-grained genetic algorithms [1, 9]. We have limited our literature review to some of the most relevant works which have used the concepts of genetic algorithm and single objective fitness function in testing. A detailed study of these works is done to make an assessment of genetic algorithm based software testing approach. In the next paragraph, we discuss some of the most relevant works in genetic algorithm based software testing.

A path wise test data generation using genetic algorithms was introduced by Pei et al. [45]. A control flow graph was constructed and the paths were manually selected from the graph. Only two loops were covered at a time. They designed the fitness function based on the paths selected from the graph. Genetic algorithm based testing was used by Roper et al. [46] for testing C program. They used the branch coverage criteria. In their approach, a random method for population selection was used and this population was subjected to crossover and mutation to generate better individuals. A branch coverage criterion was used by Jones et al. [30] in their work for generating test data using genetic algorithms. Hamming distance approach was used to design the fitness function and their approach could cover programs which contain up to three loops. Pargas et al. [44] developed a tool called TGen which uses genetic algorithm for program testing. A parallel processing approach was used in TGen to improve the testing process. A path coverage and branch coverage approach was used in TGen. The performance of TGen was compared with a tool called Random which is a tool based on random method. Test cases which covered the largest number of predicates were given the highest fitness values. Bueno et al. [7] developed a method for software testing using genetic algorithms. They used the path coverage criteria and introduced the path similarity metric as fitness function. The population initialization was made by checking the previous nature of the population. This helps to create better individuals in the successive generations. Wegener et al. [52] used a statement coverage criterion during testing and they introduced a fitness function which is decided based on the approximation level and normalized predicate level distance. Michael et al. [2001] developed a tool called GADGET which uses genetic algorithms for generating test data for C programs. They designed the fitness function based on some predicate function. Their tool had many limitations like the inability to handle Boolean variables. Doungsa-ard et al. [12] used a genetic algorithm based approach to generate test data for UML state diagrams. They used the transition coverage approach and the fitness function was designed based on the number of transitions fired by the input sequence. The population initialization was made based on the nature of the previous generation individuals. Hermadi et al. [1] used a path coverage criterion to genetic test cases using genetic algorithm. The overall fitness function was a measure of aggregation of individual's fitness function. Table 5 gives a list of some of the works which is uses genetic algorithms for software testing.

A review of these works, throws up some of the pertinent issues in genetic algorithm based software testing. These factors, which play a major role in genetic algorithm based testing and influence its outcome to a significant degree, are given below:

– Population generation
– Design of fitness function
– Response time prediction
– Setting of parameters

i. *Population generation*

This includes initialization and representation of the population, strategies for population selection and the determination of population size. The population which is initialized may itself be the set of initial potential solution. The representation of population is another issue. Population can be represented as a group of 0's and 1's, as a group of integers, as decimal numbers or as characters. In some problems a tree representation is also possible. Based on the problem, appropriate method of representation is applied. Improper representation of the individual in genetic algorithms may cause unexpected variations in the final result [24, 25].

Table 5 : Summary of GA based works on software testing

WORK	COVERAGE	FITNESS FUNCTION	GA TYPE & POPULATION REPRESENTATION	POPULATION SIZE & SELECTION STRATEGY	CROSS OVER TYPE	MUTATION TYPE
DOUNGSA-ARD et al. [2002]	Transition	Number of transitions fired by input sequence	Simple GA & Sequence of triggers	10 & Previous knowledge	Two point	Random mutation & 0.5
HERMADI et al. [2001]	Path	Fitness= Number of violations +Distance	Simple GA &	30 & Roulette wheel selection	Single point	0.1 0r 0.3
WEGENER et al. [2001]	Statement	Approximation level and normalized predicate level distance	Simple & multi population GA & Integer representation	Stochastic universal sampling	Single point	Discrete recombination, 1 & multiple strategies
BUENO et al. [2002]	Path	FT=NC-EP/MEP	Simple GA& Binary string	80 and Selection based on Previous knowledge	Single point	Simple & 0.03
MICHAEL et al. [2001]	Branch	Predicate function	Simple GA & Binary String	24, 100 and Random selection	Single point	Simple & 0.001
PRAGAS et al. [1999]	Statement & Branch	Common predicates	Simple GA & Input data list	100 & Random selection	Single point	Simple & 0.10
JONES et al. [1996]	Branch(Maximum 3 loops)	Hamming distance	Simple GA & Binary plus sign & gray code	45 & Random selection	Uniform	Reciprocal &Weighted. Reciprocal & five least
ROPER et al. [1995]	Branch	Coverage percentage	Simple GA & Character string	User input & Random selection	Single point	Simple mutation. Mutation rate decide by user

PIE et al. [1999]	Path(Maximum 2 loops)	F= C-[10*n+5*n(n-1)/2	Simple GA & Binary string	Program's s size & Random selection	Single point	Simple mutation & 0.001

The next major concerns related to population are the population selection strategy and population size. Either a random method or a heuristic based method is used to initialize the population. In the random method, population is selected randomly. In the heuristic based approach, instead of setting the population randomly, some tests are performed and the individuals are selected based on the test results. This shows that, population selection strategy can be based on several methods to select the appropriate population. The population size can also be a confounding factor because if the population size is too small the genetic algorithm will not search all the possible solution areas to procure an optimal solution [9, 12]. In this case, the individuals may reproduce abundantly and the resulting diversity in population may cause the individuals to converge to a point which appears to be better than the neighboring points. In such a situation, even though there is a chance that a better solution exists, it is missed as the population size is already declared to be very small. This is known as the premature convergence problem [40]. Hence declaring the correct population size still remains a problem in genetic algorithm and research is still ongoing in this area. Before using genetic algorithm for software testing, these inherent issues have to be addressed. Due to the shortcomings of single population genetic algorithm, parallel genetic algorithm has been tried in many applications [30]. Parallel genetic algorithms are similar to single population genetic algorithms running in different machines. The performance of parallel genetic algorithms is affected by the way in which the computers are networked. In effect, even though parallel genetic algorithms may speed up the computation process compared to single population genetic algorithm, several issues in the network implementation topology needs to be dealt with.

We have used the Genetic algorithm solver tool in Matlab 7.8 to give an idea of the population initialization issues presented above. The initial parameter settings for the Genetic algorithm tool are given below.

Genetic algorithm parameter settings
Minimize Objective Function (f) = 10-x
Population type: Double vector
Fitness Scaling: Rank
Crossover type: Scattered
Migration rate: 0.2
Stopping Criteria: 100 generations
Time limit: 10 second

All the parameters except the population size are kept constant. The result obtained for various population sizes is given in Table 6. The objective function value and the value of the best individual present in all iterations are also displayed. From Table 6, it can be inferred that as the population size increases, the result obtained becomes better. Another illustration is given below in figures 2 to 6. These show that, as the population size increases beyond a certain size, the time taken for fitness function optimization increases.

Table 6 : Function values and final point values

Populatio n Size	Best Individual final point value	Objective function value
20	24.76	-14.76
20	24.88	-14.85
30	27.30	-17.30
30	28.37	-18.37
70	44.51	-34.51
70	39.95	-29.95
1000	67.99	-57.99

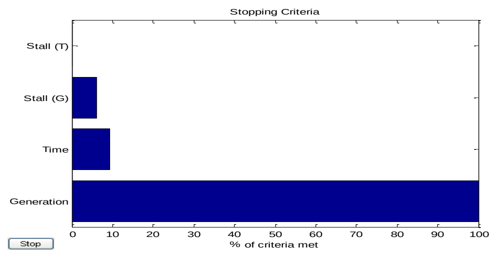

Figure 2 : Stopping criteria for population size =20

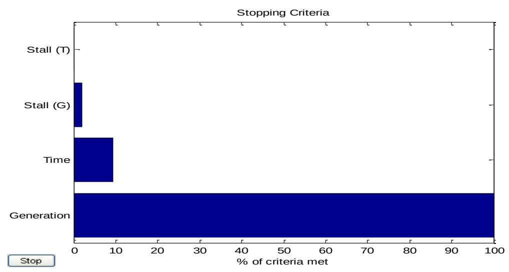

Figure 3 : Stopping criteria for population size =30

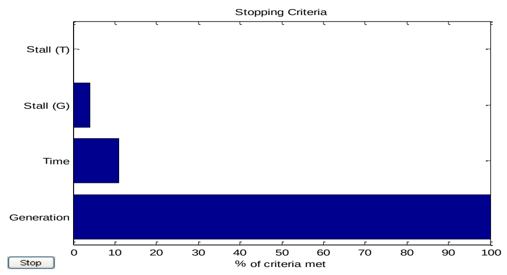

Figure 4 : Stopping criteria for population size =70

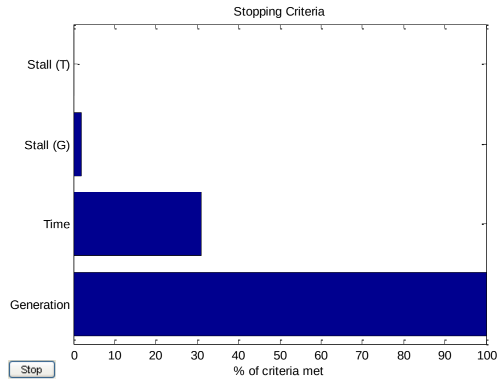

Figure 5 : Stopping criteria for population size =1000

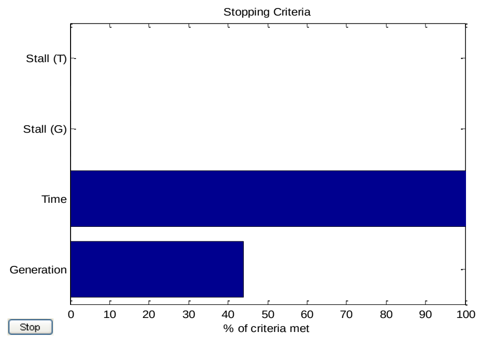

Figure 6 : Stopping criteria for population size =10000

In figures 2 to 6, the time taken to compute the fitness function optimization process for population sizes 20, 30, 70, 1000 and 10000 is showed. Some of the terms related to the fields in these figures are explained below.

- Stopping criteria: Decides the cause of algorithm termination
- Generations: Gives the maximum number of iteration the genetic algorithm runs before termination
- Time limit: Gives the maximum time limit in seconds the genetic algorithm should function before termination
- Stall generations: Genetic algorithm terminates when the weighted average change in the fitness function value over stall generations is less than function tolerance
- Stall time limit: Genetic algorithm terminates when there is no improvement in the fitness value which is the best in a specified time interval
- Function tolerance: The algorithm stops if the weighted average relative change in the best fitness function value over Stall generations is less than or equal to Function tolerance.

When the population size is defined as 20, 30 and 70 respectively, the corresponding fitness values are obtained and the genetic algorithm terminates when the maximum number of generations are exceeded. The time taken for these three processes is almost the same. These can be inferred from the results given in figure 2, 3 and 4. In figure 5 when the population size is 1000, the time taken for fitness function optimization is greater compared to the time taken for population size 20, 30 and 70 and here also the genetic algorithm terminates when the maximum number of generations exceeded the limit specified. In figure 6, it can be seen that only 44 iterations were able to run within the time limit specified as the time limit exceeded the maximum value. Here, an increase population size caused an overrun in time limit. These results point out that population initialization can influence the final result and the population initialization process is problem dependent. For small non- critical optimization problems, the size of the population may not be a critical factor. In critical problems, the population size is very crucial [50].

ii. *Setting of parameters*

In genetic algorithm based program testing, the parameter setting needs special attention. For example in the case of crossover and mutation, their rates should be not be set at either high or low levels. According to the problem's nature the parameter settings should be adjusted. The following section gives a description of some of the operator settings used in genetic algorithm based testing.

a. *Selection*

In selection, individuals are selected from the parent population for crossover and mutation to produce next generation individuals [28, 45, 51]. There are different types of selections like roulette wheel, tournament selection, random selection, best selection etc. In roulette wheel selection individuals are selected according to their fitness. Each individual will be assigned a fitness value and the normalized fitness value is calculated. After calculating the normalized fitness value, accumulated fitness value is calculated by adding the fitness value of the concerned individual and the sum of the fitness value of all other individuals. A random number is selected between 0 and 1 and the selected individual will have an accumulated fitness value greater than all other previous individuals but less than the remaining individuals. Tournament selection is a refinement of roulette wheel selection. Here roulette wheel selection is repeatedly applied to produce a group of population and the best individual is selected from this group. In random selection method, the chromosome is selected randomly from the given population whereas in best selection method the individual with the highest fitness value is selected. There are many other types of selection methods, but we have mentioned only a few. There is no specific rule which implies the usage of a particular type of selection method during software testing process. This is one of the greatest difficulties in genetic algorithm based software testing, as the final outcome of testing differs according to the type of selection method used.

b. Crossover

Crossover is the process of combination of parent chromosomes to produce offspring [HOLLAND 1979]. The process of crossover affects the process of test data generation using single population genetic algorithm. The most commonly used types of crossover are one point crossover, two point crossover and uniform crossover. For example consider two parent individuals where the chromosomes are represented as bit strings:

Parent 1:1010101010
Parent 2:1000110000

If the crossover occurs after the sixth bit in the parents, then two children will be formed and the last four bits of both the parents are interchanged. The result can be represented as follows:

Child 1:1010100000
Child 2:1000111010

In uniform crossover, the crossover points are not selected. The parent bits are swapped randomly with 50% probability. If the third, sixth, seventh and tenth bit positions of the parent individuals are swapped, then two children will be produced and they can be represented as follows:

Child 1:1000110010
Child 2:1010101000

By using uniform crossover the diversity in the individuals produced is more compared to single and two point crossover and a better result is obtained. A better result for a given problem may be obtained, even if the testing process is done with the most suitable type of crossover. Solving this uncertainty in genetic algorithm crossover selection still remains as a challenge.

c. Mutation

Mutation is the process of altering the value of genes present in the chromosome for creating genetic diversity [18]. Diversity in the population will create better individuals compared to a population without genetic diversity. According to the problem to be solved, mutation rates can be set to specific values. If the rate of mutation is set to high value, the search will become similar to a random search and if the mutation rate is very low then there will be no diversity in the population. Therefore generally the value of mutation is set between 0.01 and 0.05 [40]. From table 5, we can notice that the mutation rate is set to different values in the listed works. The main problem faced here is that, varying the mutation rate results in a change in the final result and this issue still remains unresolved in genetic algorithm based testing process.

d. Uncertainty in Parameter Settings

Even after testing a program using the best available genetic parameters, a better solution or the same solution can be obtained even if we use less competing methods of crossover, selection and mutation for solving the same problem. This shows the uncertain nature of genetic algorithms [38]. We have some examples to illustrate the uncertainty of genetic algorithms. Our aim is to use genetic algorithms to minimize the SchafferF6 function, which is a published benchmark function. SchafferF6 function is a complex optimization problem whose solution can be obtained by applying genetic algorithm based optimization methods. We have considered SchafferF6 function in our optimization test because this function is a multidimensional function. It is having non-linear and oscillatory nature around the optimal solution [18]. This means that SchafferF6 function is having more than single local optima where the genetic algorithm may get halted.

The SchafferF6 function is defined as:

$$f(x) = 0.5 + \frac{(\sin^2\sqrt{x^2+y^2}) - 0.5}{(1.0+0.001(x^2+y^2))^2}$$

Here function minimization is done using two-point crossover and uniform crossover. Initially the objective function or the fitness function minimization is done using two-point crossover. Then the experiment is repeated again using the same parameter settings. The resultant values are noted in each case. Then the objective function minimization is done using uniform crossover. Here also the experiment is repeated using uniform crossover and the values are noted. The results are shown in the table 7, table 8 and table 9.

It has been said that when uniform crossover is used for solving a problem, not only the result will be better compared to two point crossover, but also the convergence happens faster [30]. From the illustrations given below in tables 7, 8, 9, we can see that this is not true in all the cases. In the first trial, the value of the

SchafferF6 function obtained using uniform crossover is better than two point crossover. Further in this case, the time taken is more compared to two-point crossover. In the second trial, the time taken for minimizing the objective function using uniform crossover is less compared to two-point crossover. Here we can see that the fitness function has lower value when two-point crossover is used. Even though there is a little bit difference in time taken to minimize the function, the quality of the result is better in two- point crossover. In the third trial also the value of the fitness function is better when two-point crossover is used. Here the time take is more when two point crossover is used. When uniform crossover is used in the third trial it can be noticed that the value of the fitness function is greater than the value of the fitness function got when two point crossover is used and this indicates that the quality of test data got using two point crossover is better than the quality of test data obtained using uniform crossover. We can see that the time taken for two-point crossover is more compared to time taken when uniform crossover is used. Even though the convergence takes place faster in uniform crossover, it is not mandatory to get minimal value of the fitness function in all the trials.

Table 7 : Trial 1 values

Parameters	Two-Point Crossover	Uniform crossover
Number of Generations	1070	3184
Time taken in seconds	7.657	26.649
Score	0.001982	0.001758
Fitness function Value	0.265497	0.198465

Table 8 : Trial 2 values

Parameters	Two-Point Crossover	Uniform crossover
Number of Generations	949	749
Time taken in seconds	7.336	6.258
Score	0.003094	0.000808
Fitness function Value	0.257263	0.362636

Table 9 : Trial 1 values

Parameters	Two-Point Crossover	Uniform crossover
Number of Generations	499	145
Time taken in seconds	4.543	1.010
Score	0.001609	0.001124
Fitness function Value	0.167003	0.332225

Form these observations we can conclude that, even though there are some general assumptions about the best methods of crossover, selection and mutation, which are to be used for solving a problem, it may not be possible to decide the best combination of these genetic factors as parameter setting in all the cases [26]. Therefore while using genetic algorithms for program testing; we can make only a few assumptions about the problem which is to be solved. All these make the use of genetic algorithm for effective program testing highly complex and impractical.

iii. *Design of fitness function*

Applying genetic algorithm in program testing requires optimizing the specified fitness function. A fitness function should be designed in such a way that it gives optimal solution for a given problem. Defining the fitness function imprecisely may lead to a wrong solution or may cause the problem to be stuck in the local optima [18, 40]. The misleading nature of fitness function creates several problems. For example, the individuals with lower fitness values may be finalized as the optimal solution even when better individuals exist. This mainly occurs when the population size is smaller, because with a small sized population, the result may get converged at a faster rate than normal. Thus, in a limited population, if one of the individuals surpasses the neighbouring individuals, then that point or individual will be considered as the best solution even when better solutions exist. Considering these local points as the candidate solutions and assigning higher fitness values to them will result in a diversion from the original solution. This results from the inherent weakness of genetic algorithms [40]. A group of researchers used an evolutionary algorithm along with a reprogrammable hardware array and the fitness function was designed to output an oscillating signal. At the final stage of the experiment, the researchers found that the circuit had become a radio receiver which was able to pick up and relay an oscillating signal from the nearby electronic device. Here, there was a deviation from the main goal itself and this was due to the fault in the design of the fitness function [19]. Each one of the many works which use genetic algorithm for software testing has designed their own fitness function [37]. Referring the works given in table 5, we can see that none of the works have used similar type of fitness function. For example, Bueno et al. [7] have used a path similarity metric as fitness function and Michael et al. [40] have used the fitness function based on some predicate function. Even though there are some good methods for fitness function calculation, none of them is universally accepted as the gold standard. The fitness function is designed based on the analysis of a problem [24]. In other words, fitness function is problem dependent and this is one of the hurdles to be surmounted while using a genetic approach in software testing.

iv. *Response time prediction*

Fitness function optimization is a heuristic process and the optimization time and effort varies according to the nature of the problem [2]. Therefore, the exact time required for testing a program cannot be accurately predicted. The time varies as the parameter settings are changed. These can be inferred from the

© 2014 Global Journals Inc. (US)

graphical figures 2 to 6. From these figures, it is clear that solving a problem with a lower population size will take less time compared to solving the same problem with a higher population size. Even though this is not a major concern in most of the testing applications, some care has to be taken while using genetic algorithm based testing in safety critical applications. In today's world, the workings of all applications are based on real time software. In real-time system the response time plays a critical role and due to the long computation time and uncertainty in the duration of computation time, genetic algorithms cannot ensure constant response time in all the executions [50]. Therefore before implementing the genetic algorithm based system in the original system, a prototype model checking has to be carried out. As stated above, since the performance of genetic algorithm changes according to the change in the parameter values, using genetic algorithms to solve such real time problems should be done with utmost care.

c) *Software Testing using Program Slicing*

In the previous section we saw an overview of genetic algorithm based software testing. We have also explained some issues which can make genetic algorithm based software testing less practical in testing industry. This section looks into the possibilities of program slicing for software testing.

As mentioned in section 2, the concept of slicing was introduced by Weiser and his works encouraged the application of slicing in several fields like program comprehension [22], testing [20, 21, 47], debugging [33, 34], software maintenance [16], program cohesion [43], refactoring [35], reverse engineering [8] etc. We shall see how it can be used for software testing. In software testing, locating the erroneous statements is the key part. As program slicing deletes all those statements from a program which cannot affect the values of a variable of interest, slicing can make the whole software testing process more manageable. Even though some works have mentioned the use of slicing in testing [3, 5, 20, 21, 47], work that has explicitly shown how program slicing may be applied in software testing is extremely rare to the best of our knowledge. We have mentioned some fundamental works in table 10 which apply slicing for identifying test cases during the various phases of software development life cycle. In these works we can notice that they have either mentioned the need of slicing during regression testing process or during the design phase for identifying test cases before the coding phase. Our work illustrates how test cases may be obtained from slices during the testing phase itself.

Table 10 : Works on Program slicing based software testing

Work	Description
Gupta et. al[1992]	Regression testing using slicing
Binkley[1998]	Incremental regression testing using slicing
Harman et al. [1994]	Mentioned that slicing may be applied during the testing phase by checking whether the program is robust or not
Bates et al.[1993]	Slicing applied to identify statements modified in a program dependence graph during the regression testing phase
Samuel et. al[2009]	Using dynamic slicing to generate test cases form UML activity diagrams

i. *Testing Approach*

We have used a forward slicing approach for program testing. Forward slicing is recommended to locate the parts of the program affected by some modification and the sizes of the forward slices are smaller than that of backward slices in some scenarios [22]. In other words, when testing is done with an aim of identifying the errors caused by wrong input variable declaration, forward slicing is more meaningful than static slicing [22]. If the user is supposed to find errors in the output variable then static slicing is more useful than dynamic slicing. In such scenarios it will be more meaningful to apply forward slicing rather than backward slicing. In forward slicing, if a particular statement is affected by the value of the slice variable which is declared at a particular point, then that statement can be added to the list of slice statements. Otherwise there is no need to update the slice list. The whole process will be continued until slicing is performed for all the required variables. The result of the whole process will be a set of statements. These statements are known as forward slice of a particular variable. The forward slicing algorithm suggested in this work is given in algorithm 2

ALGORITHM 2

```
Input: - Program to be sliced (P)
Slicing Criterion (C)
Output: - Forward Slices (F)
begin
1. while p ≠ Ø, source program not empty
2. get C= (n, V)
// where n is statement number, V is the slicing variable
3. while (n ≠ 0 && n < EOP)
```

Global Journal of Computer Science and Technology (C) Volume XIV Issue VIII Version I Year 2014

```
//where EOP is the end of program
{
4. Store 'V' in 'L'
// slicing variable 'V' stored in list 'L'
5. if ( VAR (L) Є n)
// check whether slice variable 'V' stored in list 'L' is present in statement 'n'
{
5.1. if ( n is an element of output statement)
F= F U n
//Store n
// include the statement as a slice
5.2. if (n is an element expression)
{
5.2.1. if (VAR (L) Є RHS (EXPR))
{
```

```
F=F U n
// Store n
VAR (L) = VAR (L) U VAR (LHS (EXPR))
}
// include the statement as a slice
// VAR (L) is the slice variable 'V' stored in list ' L' and RHS (EXPR) denotes the right side of the expression and LHS (EXPR)
denotes the left side of the expression and VAR (RHS (EXPR)) denotes variables in the right side of the expression and VAR (LHS
(EXPR)) denote the variables in the left side of the expression.
5.2.2. else
do not include the statement as a slice
5.3. if (n is an element of a conditional statement)
{
5.3.1. if ((VAR (L) Є LHS (EXPR) ¦ ¦ (VAR (L) Є RHS (EXPR))
{
F= F U n //Store n
F= F U Loop body statements // Include all statements inside the conditional loop in F
}
}
5.4. if (n is an element of input statement)
F= F U n // include statement as a slice
5.5. if (n is an element of initialization or declaration statement)
F= F U n // include statement as a partial slice
}
6. else
n= n + 1
7. Repeat steps 5…6 until all the program statements are covered or till the EOP is reached
end
```

In the algorithm 2 given above, the user selects the program for which the test sequence is to be generated. The slicing criterion is verified initially. Slicing criterion contains the variable and statement number. Here, we have to check for the program statements that are affected by the value of a particular variable at a particular point. The slice variable 'V' is stored in a list 'L'. The statement number is denoted by 'n'. The process starts from the (nth) line till the end the program is reached. In the (nth) line, it is checked whether the variable 'V' is present or not. If the variable 'V' is not present, then (n+1) th line is checked. If the variable 'V' is present in the (n) the line, a series of steps are to be performed. If 'V' is present in an expression, it is checked whether 'V' is present on the right side or left side of the expression. If 'V' is on the left side of the expression, that statement is considered as a slice and all the variables in the right side of the expression are also added to the list. In 'V' is in the right side then it is not included as a slice. While checking the next line, we have to check not only for 'V', but also all the all the variables present in the list. This is because; the other variables added to the list are the dependent variables of 'V'. Similarly, it is checked whether the slice variable is an element of conditional statement, declaration statement, input statement and output statement. If these conditions are true, the statements are considered as a slice. The statements inside the conditional body loop are also included as slice because the executions of these statements are dependent on the conditional clause. The process is repeated unit the end of the program and the result will be the forward slice for the corresponding

ii. System Description

An overview of our system model is given below. Our system is implemented using Java and Netbeans IDE. Netbeans is having extensible plug-in system and Java is having object-oriented features. This is why they have been used. The main modules of the implemented system consist of the following parts, given in figure 7.

1. Input unit
2. Slicer
3. Analyzer and tester

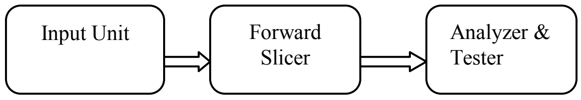

Figure 7: Main modules of slicing based system

a. *Input unit*

The input unit has the facility to select the software program which is to be tested. After selecting the program, the variables in the program are listed. From the listed variables, the user can select the variables for slicing criterion.

b. *Forward Slicer*

This is the main part of the system. In this unit, slicing is performed for the program which is to be tested. After getting the program and the list of variables from the input unit, forward slicing is performed to identify the relevant statements in the selected program with respect to the slicing criterion. Forward slicing is performed according to algorithm 2 given in section 3.3.1. A sample program code is given in Sample 1 and the working of forward slicing algorithm is explained below. In the program code given above in Sample 1, forward slicing is applied with respect to the input variable ‘basic‘. The slicing criterion given is C= (3, basic). The result of forward slicing is given in Result 1.

Sample 1

```
1. main( )
2. {
3. float basic, total, da, rent;
4. if (basic < 1000)
5. {
6. rent= basic * 12 /100;
7. da= basic * 60 / 100;
8.}
9. else
10. {
11. rent= 700;
12. da= basic * 80 / 100;
13.}
14. total =basic + rent + da;
15. System .out. println (—total = —+ total);
16.}
```

Result 1

```
4. if (basic < 1000)
6. rent= basic * 12 /100;
7. da= basic * 60 / 100;
9. else
11. rent= 700;
12. da= basic * 80 / 100;
14. total =basic + rent + da;
15. System .out. println (—total = —+ total);
```

The slicer will analyze the statements 4- 16 in Sample 1. Here statements 4, 6, 7, 9, 11, 12, 14, 15 will execute based on the value substituted for the variable ‘basic‘. We can notice that the dependencies are checked in a forward direction. The final value of variables ‘rent‘, ‘da‘ and ‘total‘ are dependent on ‘basic‘. Thus forward slices obtained can find if any errors are present in the dependent statements also. The resultant statements from forward slicing are given in Result 1.

c. *Analyzer & Tester*

In this unit the forward slices obtained are verified to find out whether they are significant in testing or not. Among the forward slices given above in Result 1, these statements are relevant in testing.

Testing using Slicing

```
4. if (basic < 1000)
6. rent= basic * 12 /100;
7. da= basic * 60 / 100;
9. else
11. rent= 700;
12. da= basic * 80 / 100;
14. total =basic + rent + da;
15. System .out. println (—total = —+ total);
```

The execution of the rest of the program statements is dependent on the value of the variable ‘basic‘. Here the tester identifies the test sequence statements which are relevant for generating the required test data values from the forward slices. In order to find the possible value of ‘basic‘ present in the conditional statement of the static slice, an equivalence partition method is applied. Equivalence partition is considered as the basis of all testing data generation methods and in this method, when a program works for a particular value in a partition, it may work for the other values in the same partition and this in turn helps to avoid duplicate testing [31]. Moreover, equivalence partition method is comparatively easy and reliable [31]. In equivalence partition, the input domain is divided into a number of sub domains. The sub domains make up the equivalence class. If a test data value in a class or partition is considered as a right value, then all the values under that particular class is considered as good values. We have to generate a value for the variable ‘basic‘ using equivalence partition. From the slice given in this section, conditional constraint is given is ‘if (basic<1000)‘. Here the possible partitions are (basic >1000)‘. and (basic >1000)‘. Using these partitions values are generated, which are given in table 11.

Global Journal of Computer Science and Technology (C) Volume XIV Issue VIII Version I Year 2014

© 2014 Global Journals Inc. (US)

Table 11 : Valid and invalid test data values

Partition	Test Values	Result
> 1000	1500 1010 2000	Invalid
< 1000	800 900 700	Valid

From table 11, some of the valid and invalid test data values for the clause 'if (basic < 1000)' is obtained. The invalid test values is applicable to the 'else' part of the conditional clause 'if (basic < 1000)'. Substituting some of the test data values of 'basic' in the expressions will give the value of 'da' and 'rent' and finally the value of 'total' may be calculated from these data.

IV. Research Approach

In the previous sections, we have analysed program testing using genetic algorithm and program slicing methods. Some issues related to genetic algorithm based testing have also been pointed out. Based on these observations, we have framed some research questions (Q) in the coming section. The aims of the research questions are also mentioned and this may help future research work in this area.

a) *Research Questions*

Q1. What is the future of genetic algorithm based software testing?

The aim of this research question is to analyze the effectiveness of genetic algorithm based software testing. This question also intends to deal with the practical difficulties of this type of testing.

Q2. In the software testing context, why is program slicing considered a better approach?

This question aims to analyze the strengths of program slicing in testing and to study how program slicing makes testing more effective and reliable.

b) *Review Method*

We have referred to some relevant works in the field of genetic algorithm and program slicing based testing. A lot of works use genetic algorithms for test selection, test prioritisation, hardware testing etc. Apart from this, several works use a combined approach which uses genetic algorithm and other search algorithms for software testing [9]. Here we have mentioned only those works that describe software testing and test data generation using single population genetic algorithm. We have not considered other variations of genetic algorithms like parallel genetic algorithm as they are not employed in testing literature. We have reviewed several papers which describe slicing concepts, various types of slicing, slicing algorithms, applications of slicing etc. None of them have mentioned how to proceed to the testing phase after obtaining the slices. As our focus in on program slicing based software testing, we have selected some leading works which have mentioned the term 'testing' along with program slicing which is listed in table 10. Also, we have considered some of the fundamental works which use genetic algorithms for test case generation. We have not considered test selection, prioritisation etc. A summary of the referred works are given in table 5. The study made in section 3.2 answers the research questions.

V. Results

In this section we have tried to give an explanation to the research questions based on the studies mentioned in the previous sections.

Q1. What is the future of genetic algorithm based software testing?

We have provided only the most relevant points as solution to the research question. For this, the question Q1 has been split into some secondary questions (SQ). Providing appropriate answers to the secondary questions leads to an unbiased review of genetic algorithm based testing.

SQ1. What is the role of genetic operators in genetic algorithm based testing?

All the reviewed works use only single point crossover, except Jones et al. [30] work. In Jones's et al. [30] work, uniform crossover is used. Also, while others use simple mutation and Jones's work uses reciprocal and weighted mutation. Even though several works which explain the different types of operators and their relevance in different contexts exist, none of them have exploited these operators. They have used only the direct type of operators in their work. All these show that, the result obtained by using these common types of operators may be improved by substituting the testing process with a general operator selection strategy. This has not been decided till now in genetic algorithm based testing.

SQ2. Does population initialization and representation affect software testing?

From section 3.2, we can see that the population is selected randomly in most of the works. Selecting the population based on some heuristics improves the software testing process. Apart from this, we can see that only single population is used in most of these works. Only Wegner's et al. [52] work use multi-population along with single population. Even though a lot of research works are conducted continuously to decide the best type of population initialization, selection etc., some of the most common works which used genetic algorithm for software testing have experimented very little with population initialization methods. Again this shows that the quality of genetic algorithm based testing is dependent on population initialization and the lack of a general strategy for population setting makes the whole testing process unpredictable.

SQ3. What are the problems related with fitness function design during software testing?

Applying genetic algorithm in program testing requires optimizing the specified fitness function. A fitness function should be designed in such a way that it gives optimal solution for a given problem. Defining the fitness function imprecisely will lead to a wrong solution or in some cases the problem may get stuck in local optima [18, 25] suggested a method to remove variables which can lead to local optima. Even though, they were able to alleviate the problem of local optima, their approach didn't work for inner loop variables. Another problem faced during the fitness function design process is the dependency problem. While designing the fitness function for a target node, the dependent nodes which affect the target node should be considered. Since most of the works, which use genetic algorithm based approach for testing, do not use data flow criteria, the fitness value may not be correct. Some works were done on this area to minimize this problem, but they could not explain the best strategy for fitness function design in the context of testing [26, 50].

SQ4. Program dependency

In most of the genetic algorithm based software testing, program dependency is not correctly followed [37, 24]. In genetic algorithm based program testing, initially all the statements in the program should be analyzed to identify the relevant statements or we have to get the list of statements that will have a potential role in software testing. From the testing point of view, checking the whole program line by line is an unnecessary waste of effort. Instead of that, if we are able to find the program statements which help in program testing, such as those that assist in finding the test data values during testing, the whole testing effort will be reduced considerably. In addition, the testing can be made more methodical. Identifying the relevant statements which contribute to program testing, and analyzing those statements can give the dependence relation present in the program. Utilizing this dependence relation helps to trace out the errors in a program. For example, consider the sample control flow graph given below in Figure 8. All the program statements will be checked line by line from the starting point of the program. The statement 'basic<1000' assist in test data generation and suitable test data values should be generated for the variable basic'. The value of 'basic' is found out by optimizing the function f(x) = 1000 – basic. After finding out suitable values for the variable 'basic', the successive statements in the program is checked for errors. This is how the testing proceeds in this approach. In order to get a full satisfactory explanation for SQ4, we have to see the result research question Q2. The explanation given in Q2 provides a justification for SQ4.

Q2. In the software testing context, why is program slicing considered a better approach?

In the above section we saw some of the shortcomings of genetic algorithm based testing approach. An example given below gives an explanation to research question Q2. Consider the same example given in figure 8. In the control flow graph, the statements which correspond to each node are marked. From the control flow graph we are taking the forward slicing criterion as (2, basic). This means that all the statements which are affected by declaring the variable 'basic' in statement 2 is to be identified. The resultant nodes in the CFG are given below in Figure 9.

© 2014 Global Journals Inc. (US)

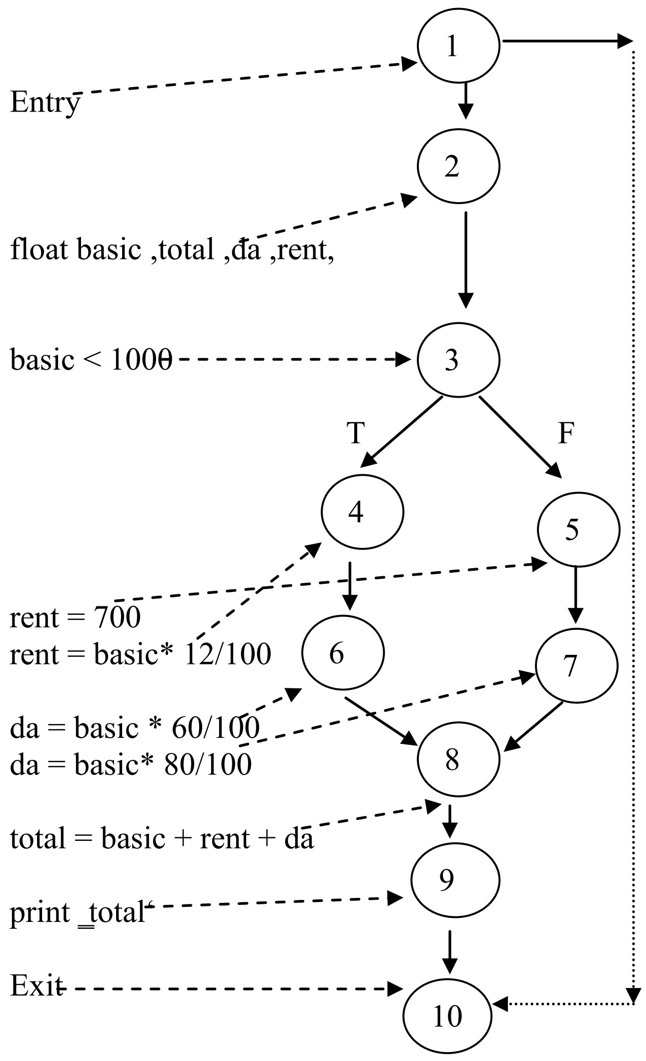

Figure 8 : Sample Control flow graph

It can be observed that all nodes displayed above will be affected by the variable 'basic' in statement 2. Node 3 is given as (basic<1000). When this program is to be tested, the test data which satisfies the condition in node 3 is to be generated. Similarly, nodes 4 and 6 are dependent on node 3 and this can be clearly traced form the slices obtained. Nodes 5 and 7 are also dependent on the variable 'basic'. If the value of 'basic' is greater than 1000, then these nodes get executed. From this we can conclude that the statements which are relevant in testing and in the successive stages of testing like test case generation can be identified easily by the process of slicing. Moreover, as slicing gives the dependence information present in a program, it will be easy to dig up the mistakes in the dependent statements.

© 2014 Global Journals Inc. (US)

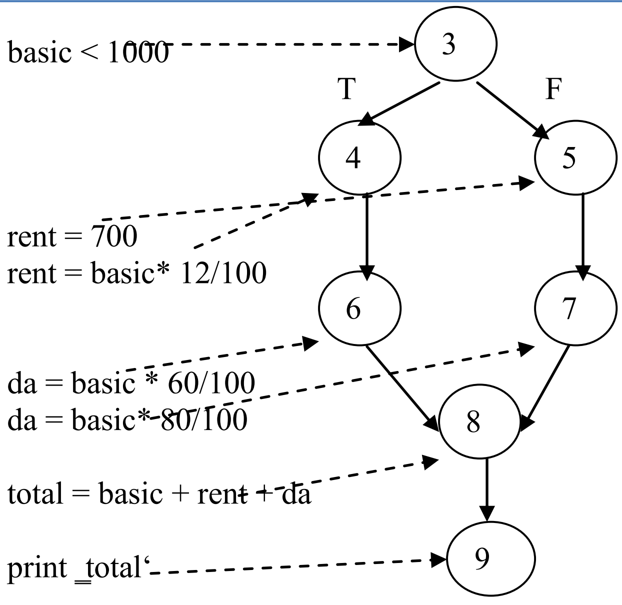

Figure 9 : Control flow graph obtained for slicing criterion (2, basic)

We saw that, for testing the same program given in figure 8, if genetic algorithm is used instead of program slicing, the program statements will be checked line by line from the starting point of the program. The main difficulty in this approach is that all the statements which contain relevant and irrelevant variables should be analyzed to trace the errors in the program code. On the other hand, as program slicing is done based on some slicing criterion, an overview of the dependence in the program code is revealed and error detection will be much easier. Here we can notice that every input variable present in a program will not be responsible for the execution of branches present in the program. Moreover, removing the irrelevant variables from a program and focusing only on the relevant variables which are significant in the execution of a target branch can improve the performance of genetic algorithm based testing. Relevant variables are those which can influence certain statements in a program, while irrelevant variables are those that cannot affect the program statements. This points out the fact that, genetic algorithm may not perform up to the mark in a practical program testing scenario [39], which underscores the superiority of program slicing in program testing. A graph is given in Figure 10 which gives an analysis of the performance of evolutionary algorithms with and without irrelevant variable removal. Here in y-axis the success rate is plotted and in x-axis the program names with branches are plotted. Here P1 denotes the program name, F1 denotes the function and B1, B2 and B3 denote different branches. Success rate is a measure of optimal test cases found out for the program branches. It can be noticed that the performance is better when irrelevant variables are removed from a program, compared to the performance without irrelevant variable removal. This establishes the weakness of genetic algorithm when there are a large number of irrelevant variables.

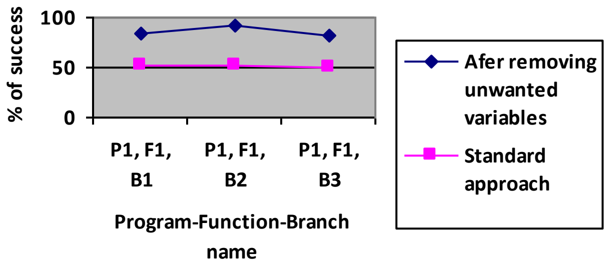

Figure 10 : Performance of evolutionary algorithms with and without irrelevant variable removal

Our observations, which are listed below, add more weight to the research question Q2. We have done an analysis of the number of program statements which have a significant role in program testing identified by both program slicing based testing and genetic algorithm based testing. The statements in a program have been considered as a metric for analyzing both these approaches to program testing. For a given program which is to be tested, the forward slicing covers more number of program statements compared to genetic algorithm in the same time span with respect to a particular variable. As the probability of error distribution in a program is uniform throughout the code, an increase in the number of executable statements with respect to a particular program variable increases the chance of discovering the number of faults related to that variable [33]. This means that, rather than concentrating on a particular area for a long time to attain high coverage for that particular branch or program code, program slicing tries to analyze more number of potential statements in a given program. Here the main principle is to identify possible program statements due to which program malfunctioning is caused, using minimal testing. This re-affirms the fact that program slicing can be more effective in program testing compared to genetic algorithm.

An assessment of testing productivity obtained in genetic algorithm and program slicing based testing approach is given in figure 11.

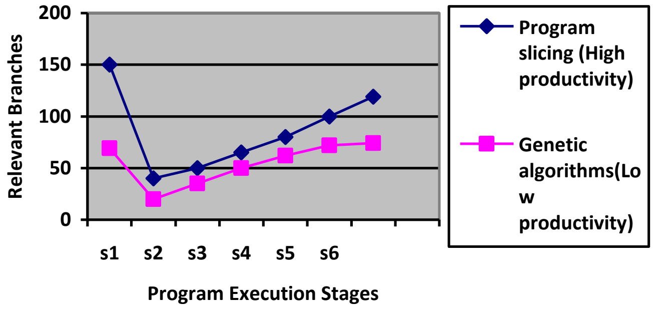

Figure 11 : Productivity graph

The graph shows that, when program testing is done using program slicing, there will be high testing productivity and when program testing is implemented using genetic algorithms, the testing productivity will be low. Some of the terms related to the graph in Figure 11 are given below.

1. Relevant branch indicates the statements of a program which may play a critical role in program testing.
2. Testing productivity indicates the measure of the number of relevant statements that can be covered in a specific time interval [31, 4].
3. High testing productivity means that more errors can be detected with less 'effort', while low testing productivity means that the number of relevant statements covered in a specific time interval will be very few [31,4].
4. 'Effort' means the time taken to detect the potential statements which contribute in program test data generation, run the program with the generated test cases and add the test cases to the test suit.

In program testing, the main objective is to find the maximum number of errors in the minimum time duration. Program slicing identifies more number of errors in less amount of time during the initial program execution stage. The relevant statements identified by program slicing provide an overview of dependency present in the program, making the error detection more practical. From this it is clear that, in program slicing based testing, although it is not possible to cover all the potential statements useful for testing, a reasonable number of statements can be analyzed when compared to genetic algorithm based program testing.

VI. Threats To Validity

The main threat to the validity of our work may be due to the limitation in the number and scope of the works which we have referred. We have limited our analysis to only those works which have mentioned the application of genetic algorithm in software testing and the use of program slicing in software testing.

The downside of such restriction in the selection of works was that, all the possible variants of genetic algorithm based testing have not been analysed. Also, we have not studied all the existing algorithms in program slicing which may have some relevance in the field of software testing. Our study has been limited to only those works which have explicitly mentioned the use of program slicing in testing. We feel that such a narrowing in the field of our study has sharpened its focus and enabled us to do an in depth analysis of our chosen study objectives; which being the identification of shortcomings of genetic algorithm and establishing the usefulness of program slicing in practical software testing.

VII. Conclusions

The unresolved issues in practical software testing constitute the Achilles' heel of software industry. As genetic algorithm is one of the most widely used and highly regarded approaches for software testing among researchers, it is high time that we explore its critical shortcomings in practical software testing. We have made an attempt to reveal some of the difficulties due to the inherent uncertain nature of genetic algorithm based software testing. A systematic review of the works made in this study reveals that, genetic algorithm factors like fitness function, population initialization and parameter settings impact the quality of solution obtained by genetic algorithm based testing. Apart from this, we have highlighted the significance of program slicing in software testing. For a given problem, program slicing has a higher 'testing productivity' with lesser 'effort'. We have used this principle as the nidus for developing our idea. We have put forth a forward slicing based method in this work. Checking of conditional constraints in the forward slices will help to pick out the rules which are to be fulfilled when testing is carried out. We have also discussed how the dependent statements in the slices are used to trace errors during testing. Certain analytical results are also provided in our work to substantiate these facts. With this work, we intend to provide a guide to future researchers and to make software industry aware of the scope and potential of using program slicing as an effective tool in software testing. In future, we plan to elaborate upon the issues brought forth by our work which may lead to promising developments in testing field.

References Références Referencias

1. AHMED, M. A. AND HERMADI, I. 2001. GA based Test Data Generator. *In the Proceedings of Congress on Evolutionary Computation (CEC'03)*. 1, 85-91.
2. AHMED, M. A. AND HERMADI I. 2007. GA based Multiple Paths Test Data Generator. *Computer and operations research*. 35, 3107-3124.
3. BATES, S. AND HORWITZ, S.1993. Incremental Program Testing using Program Dependence Graphs. *In the Proceedings of the 20th ACM Symposium on Principles of Programming Languages*.384-396.
4. BEIZER, B. 1990. *Software Testing Techniques*, Second Edition, International Thomson Computer Press, ISBN 1-85032-880-3.
5. BINKELY, D.1998. The Application of Program Slicing to Regression Testing. Loyola College in Maryland.1-24.
6. BLACK, R. 2007. *Pragmatic Software Testing: Become an Effective & Efficient Test Professional*. John Wiley& Sons Publishers.
7. BUENO, P. M. AND JINO, S. 2002. Automatic Test Data Generation for Program Paths Using Genetic Algorithms. *International Journal of Software Engineering and Knowledge Engineering,* 12, 6, 691-709.
8. CANFORA, G., CIMITILE, A. AND DE LUCIA, A. 1998. Conditioned Program Slicing. *Information and Software Technology,* 40, 11, 595–607.
9. CHEN, Y. AND ZHONG, Y. 2008. Automatic Path-oriented Test Data Generation Using a Multi-population Genetic Algorithm. *In the Proceedings of the Fourth International Conference on Natural Computation,* China, 566-570.
10. DEMILLI, R. A. AND OFFUTT, A. J. 1991. Constraint-Based Automatic Test Data Generation. *IEEE Transactions on Software Engineering,* 17,9, 900-910
11. DE LUCIA, A. 2001. Program Slicing: Methods and Applications. *In Proceedings of the 1st IEEE Workshop on Source code Analysis and Manipulation,* 142-149.
12. DOUNGSA-ARD, C., DAHA, K., HOSSAI, A., AND SUWANNASART, T. 2002. Test Data Generation from UML State Machine Diagrams using GAs. *In Proceedings of International Conference on Software Engineering Advances*.
13. FOX, C., DANICIC, S., HARMAN, M., AND HIERONS, R. M. 2004. Con SIT: a fully automated conditioned program slicer. *Software Practice and Experience,* 34, 15–46.
14. FOX, C., HARMAN, M., HIERONS, R. M., AND DANICIC, S. 2001. Backward Conditioning: A New Program Special is ation Technique and its Application to Program Comprehension. *In 9th IEEE International Workshop on Program Comprehension,* Los Alamitos, California, USA, 89–97.
15. GALLAGHER, K. B. AND BINKLEY, D. 2008. Program Slicing. *In the Proceedings of Frontier of Software Maintenance,* 58-67.
16. GALLAGHER, K. B. AND LYLE, J. R. 1991. Using Program Slicing in Software Maintenance. *IEEE Transactions on Software Engineering,* 17, 8 (Aug.), 751–761.
17. GALLAGHER, K. B. 1990. Using Program Slicing in Software Maintenance. *Ph.D. Thesis,* University of Maryland Baltimore County.
18. GOLDBERG, D. E. 1989. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison Wesley
19. GRAHAM-ROWE, D. 2002. Radio Emerges from the Electronic Soup. *New Scientist,* 175, 2358 (Aug), 19.
20. GUPTA, R., HARROLD, M.J., AND SOFFA, M. L. 1992. An Approach to Regression Testing using Slicing. *In the Proceedings of the IEEE Conference on Software Maintenance,* Orlando, Florida, 299-308.
21. HARMAN, M. AND DANICIC, S. 1994. Using Slicing to Simplify Testing. In the Proceedings of Eurostar.
22. HARMAN, M. AND BINKLEY, D. 2005. Forward Slices are Smaller than Backward Slices. In the Proceedings of the fifth International Workshop on Source code Analysis and Manipulation, 15-24.
23. HARMAN, M. AND DANICIC, S. 1998. A New Algorithm for Slicing Unstructured Programs. Journal of Software Maintenance and Evolution 10, 6, 415–441.
24. HARMAN, M., HASSOUN, Y., LAKHOTIA, K., MCMINN, P. AND J. WEGENER, J. 2007. The Impact of Input Domain Reduction on Search-based Test Data Generation. In *ESEC-FSE '07: Proceedings of the the 6th Joint Meeting of the*

European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, New York, USA,155-164.

25. HARMAN, M., HU, L., ZHANG, X. AND MUNRO M. Side-effect Removal Transformation. 2001. *In the Proceedings of the 9th IEEE International Workshop on Program Comprehension,* Toronto, Canada, 310-319,
26. HARMAN, M., HU, L., HIERONS, R., BARESEL, A. AND STHAMER, H. 2002. Improving Evolutionary Testing by Flag Removal. *In Proceedings of the Genetic and Evolutionary Computation Conference,* New York, USA, 1359-1366.
27. Hill, T.A. 2002. Importance of Performance Stress Testing on Embedded Software Applications. *In the Proceedings of QA & Test Conference,* Spain
28. HOLLAND, J., H. 1975. *Adaptation in Natural and Artificial Systems,* University of Michigan Press, Ann Arbor.
29. HORWITZ, S., REPS, T. AND BINKLEY,D. 1988. Inter procedural Slicing using Dependence Graphs, *SIGPLAN Notices,* 23,7, 35–46.
30. JONES, B.F., STHAMER, H. H. AND EYRES, D. E. 1996. Automatic Structural Testing Using Genetic Algorithms. *Software Engineering Research Journal,* 299-306.
31. JORGENSEN, P.C.2008. *Software Testing: A Craftsman's Approach.* Auerbach Publications (Taylor and Francis group)
32. KOREL, B. 1990. Automated Software Test Data Generation, *IEEE Transactions on Software Engineering,* 16, 8,870-879.
33. KOREL, B. AND LASKI, J. 1988. Dynamic Program Slicing. *Information Processing Letters* 29, 3 (Oct.), 155–163.
34. KOREL, B. AND RILLING, J. 1998. Program Slicing in Understanding of Large Programs, In the Proceedings of the. 6th International Workshop on Program Comprehension, 145-152.
35. KOMONDOOR, R. AND HORWITZ, S. 2000. Semantics-preserving Procedure Extraction. In Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM Press, N.Y., 155–169.
36. MANTERE, T. AND ALANDER, J. T. 2005. Evolutionary Software Engineering, A Review. *Journal of Applied Soft Computing,* 5,315-331.
37. MCMINN, P. 2004. Search-based Software Test Data Generation: A Survey. *Journal of Software Testing Verification and Reliability,* 14, 2, 105-156.
38. MCMINN, P., HARMAN, M., BINKLEY, D. AND TONELLA, P. 2006. The Species per Path Approach to Search-based Test Data Generation. In the Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 06), Portland, Maine, USA, 13-24.
39. MCMINN, P., HARMAN, M., HASSOUN, Y., LAKHOTIA, K. AND WEGENER, J. 2012. Input Domain Reduction through Irrelevant Variable Removal and its Effect on Local, Global and Hybrid Search-Based Structural Test Data Generation, IEEE transactions on Software Engineering, 38, 2, 453-477.
40. MICHAEL, C. C., MCGRAW, G. E. AND SCHATZ M. A. 2001. Generating Software Test Data by Evolution. *IEEE Transactions on Software Engineering,* 27, 12 (Dec), 1085-1110.
41. MYERS, G. J. 1979. *The Art of Software Testing,* Wiley, New York.
42. OTT, L. M. AND BIEMAN, J. M. 1998. Program Slices as an Abstraction for Cohesion Measurement. *Information and Software Technology-Special issue on Program Slicing,* 40,691–700.
43. OUTT, A.J., Z. JIN, Z. AND PAN, J.1999. The Dynamic Domain Reduction approach to Test Data Generation. *Software Practice and Experience,* 29, 2, 167-193.
44. PARGAS, R. P, HARROLD, M. J. AND PECK, R. R. 1999. Test Data Generation Using Genetic Algorithms. *Journal of Software Testing, Verifications, and Reliability,* 9, 263-282.
45. PEI, M., GOODMAN, E. D, GAO, Z. AND ZHONG, K. 1994. Automated Software Test Data Generation Using A Genetic Algorithm. *Technical Report* , GARAG e of Michigan State University.
46. ROPER, M., MACLEAN, I., BROOKS, A., MILLER, J. AND WOOD, M. 1995. Genetic Algorithms and the Automatic Generation of Test Data. *Technical report* RR/95/195[EFoCS-19-95], Department of Computer Science, University of Strathclyde.
47. SAMUEL, P. AND MALL, R. 2009. Slicing based Test Case Generation from UML Activity Diagrams, ACM SIGSOFT *Software Engineering notices,* 34, 6, 1-14.
48. STHAMER, H. H. 1996. Automatic generation of Software Test Data using Genetic Algorithms, *Ph.D. Thesis,* University of Glamorgan, Pontyprid, Wales, Great Britain.
49. TIP, F. 1995. A Survey of Program Slicing Techniques, *Journal of Programming Languages,* 3, 3, 121-189.
50. TRACEY, N. 2000. A Search-Based Automated Test Data Generation Framework for Safety Critical Software. *Ph. D. thesis,* University of York.
51. WATKINS, A. 1995. A Tool for the Automatic Generation of Test Data using Genetic Algorithms. *In Proceedings of the fourth Software Quality Conference,* Dundee, Great Britain, 300-309.
52. WEGENER, J., BARESEL, A. AND STHAMER, H. 2001. Evolutionary Test Environment for Automatic Structural Testing. Journal of Information and Software Technology, 43, 841-854.

© 2014 Global Journals Inc. (US)

53. WEISER, M. Program Slicing. 1984. *IEEE Transactions on Software Engineering*. 10,4,352-357.
54. XANTHAKIS, S., ELLIS, C., SKOURLAS, C., LE GALL, A., KASTISKAS, S. AND KARAPOULIOS, K. 1992. Application of Genetic Algorithms to Software Testing. *In Proceedings of the 5th International Conference on Software Engineering and its Applications.* France,625-636.

© 2014 Global Journals Inc. (US)

This page is intentionally left blank

© 2014 Global Journals Inc. (US)