The Limitations of Genetic Algorithms in Software Testing

Sultan H. Aljahdali
College of Computers and Information Sys
Taif University, Taif, Saudi Arabia
aljahdali@tu.edu.sa

Ahmed S. Ghiduk
College of Computers and Information Sys
Taif University, Taif, Saudi Arabia
asaghiduk@tu.edu.sa

Mohammed El-Telbany
College of Computers and Information Sys
Taif University, Taif, Saudi Arabia
telbany@tu.edu.sa

Abstract—**Software test-data generation is the process of identifying a set of data, which satisfies a given testing criterion. For solving this difficult problem there were a lot of research works, which have been done in the past. The most commonly encountered are random test-data generation, symbolic test-data generation, dynamic test-data generation, and recently, test-data generation based on genetic algorithms. This paper gives a survey of the majority of software test-data generation techniques based on genetic algorithms. It compares and classifies the surveyed techniques according to the genetic algorithms features and parameters. Also, this paper shows and classifies the limitations of these techniques.**

Keywords-genetic algorithms; software testing

I. Introduction

Software testing is a main method for improving the quality and increasing the reliability of software. It is a kind of complex, labor-intensive, and time expensive work; it consumes for approximately 50% of the cost of a software system development. Increasing the degree of automation and the efficiency of software testing certainly can drop down the cost of software design, reduce the time period of software development, and increase the quality of software significantly. The critical point of the problem involved in the automation of software testing is the automation of software test-data generation. test-data generation in software testing is the process of identifying a set of program input data, which satisfies a given testing criterion. A test-data generation technique must be accompanied by an application of a test data adequacy criterion, which is a predicate that determines whether the testing process is finished. There are many test-data generation techniques such as random, symbolic, dynamic, and genetic algorithms test-data generation techniques and several test data adequacy criteria such as control flow and data flow based criteria. This paper gives a survey of the majority of software test-data generation techniques based on genetic algorithms. It attempts to compare and classifies the surveyed techniques according to the genetic algorithms features and parameters. Also, this paper shows and classifies the limitations of these techniques.

This paper is organized as follows: Section II gives a background to genetic algorithms and describes their technique to generate test data. Section III gives a review of the related test-data generation techniques, especially techniques based on genetic algorithms. Section IV presents a comparison between the surveyed test-data generation techniques based on genetic algorithms. Section V presents and classifies the limitations of these techniques. Section VI presents the conclusions and future work.

II. Genetic Algorithms

Genetic algorithms (GAs) are machine learning and optimization schemes, much like neural networks. However, genetic algorithms do not appear to suffer from local minima as badly as neural networks do. Genetic algorithms are based on the model of evolution, in which a population evolves towards overall fitness, even though individuals perish. Evolution dictates that superior individuals have a better chance of reproducing than inferior individuals, and thus are more likely to pass their superior traits on to the next generation. This "survival of the fittest" criterion was first converted to an optimization algorithm by Holland in 1975, and is today a major optimization technique for complex, nonlinear problems. In a genetic algorithm, each individual of a population is one possible solution to an optimization problem, encoded as a binary string called a chromosome. A group of these individuals will be generated, and will compete for the right to reproduce or even be carried over into the next generation of the population. Competition consists of applying a fitness function to every individual in the population; the individuals with the best result are the fittest. The next generation will then be constructed by carrying over a few of the best individuals, reproduction, and mutation. Reproduction is carried out by a "crossover" operation, similar to what happens in an animal embryo. Two chromosomes exchange portions of their code, thus forming a pair of new individuals. In the simplest form of crossover, a crossover point on the two chromosomes is selected at random, and the chromosomes exchange all data after that point, while keeping their own data up to that point. In order to introduce additional variation in the population, a mutation operator will randomly change a bit or bits in some chromosome(s). Usually, the mutation rate is kept low to permit good solutions to remain stable. The two most critical elements of a genetic algorithm are the way solutions are represented, and the fitness function, both of which are problem-dependent. The coding for a solution must be designed to represent a possibly complicated idea or sequence of steps. The fitness function must not only interpret the encoding of solutions, but also must establish a ranking of different solutions. The fitness function is what will drive the entire population of solutions towards a globally best. Figure 1 illustrates the basic steps in the canonical genetic algorithms.

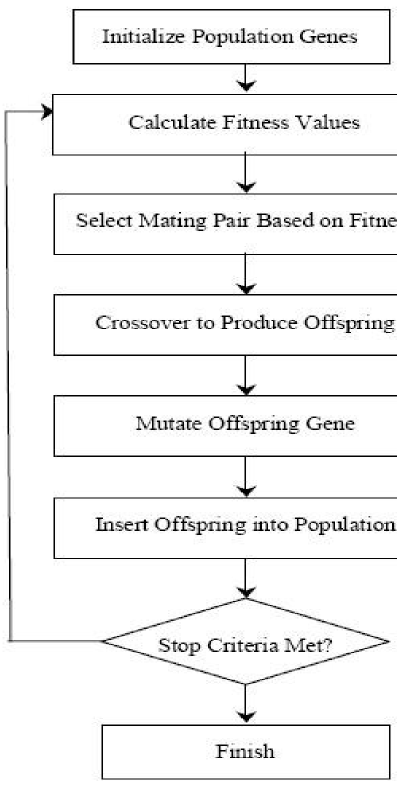
Initialize Population Genes
Calculate Fitness Values
Select Mating Pair Based on Fitness
Crossover to Produce Offspring
Mutate Offspring Gene
Insert Offspring into Population
Stop Criteria Met?
Finish

Figure 1. The canonical GA algorithm.

Using genetic algorithms in test data generation for software testing is the process of identifying a set of program input data, which satisfies a given testing criterion. In translating the concepts of genetic algorithms to the problem of test-data generation we perform the following tasks:

1. First of all we consider the population to be a set of test data.
2. Find the set of test data that represents the initial population. This set is randomly generated according to the format and type of data used by the program under test.
3. Determining the fitness of each individual which is based on a fitness function that is problem-dependent.
4. Select two individuals that will be mated to contribute to the next generation.
5. Apply the crossover and mutation processes.
6. The above algorithm will iterate until the population has evolved to form a solution to the problem (satisfies a given testing criterion), or until a termination condition is satisfied.

III. THE RELATED WORK

One of the major difficulties in software testing is the automatic generation of test data that satisfy a given adequacy criterion. To solve this difficult problem there were a lot of research works, which have been done in the past. Perhaps the most commonly encountered are random test-data generation, symbolic (or path-oriented) test-data generation, dynamic test-data generation, and recently, test-data generation based on genetic algorithms (GAs).

Recently different techniques have been proposed which are based on genetic algorithms (GAs) to generate test data. McMinn [15] and Mantere [16] survey some of the work undertaken in this field. Xanthakis et al. in [17] is presented the first work applying genetic algorithms to generate test data. In this work GAs are employed to generate test data for structures not covered by random search. A path is chosen by the user, and the relevant branch predicates are extracted from the program. The GA is then used to find input data that satisfies all branch predicates at once, with the fitness function summing branch distance values. Pei et. al. [18] presented a new approach focuses on pathwise test-data generation. Where the basic operations of pathwise software testing consist of there steps: program control flow graph construction, path selection, and test-data generation and dynamic program execution. This approach manually selects the set of paths limited to 2 loops. The overall suitability by the chromosome, that is the matching degree between the path of practical execution and the ideal required path they set, is termed its fitness. The value of fitness function of a chromosome reflects the path of the program executing on the input values of all variables represented by the chromosome how good it complies with the user selected path. Watkins [19] attempted to obtain full path coverage for programs. The fitness function penalizes individuals that follow already covered paths, by assigning a value that is the inverse of the number of times the path has already been executed during the search. The direction of the search, therefore, is under constant adaptation. However, the penalization of covered paths, in itself, provides little guidance to the discovery of new, previously unfound paths. The results show that in comparison with random testing, the GAs approach required an order of magnitude fewer tests to achieve path coverage for two experimental programs. However, both of these programs are of a simple nature, containing no loops. Furthermore, the input domains were artificially restricted for the search. Roper et. al. [20] described a system developed to explore the use of GAs to generate test data to automatically satisfy branch coverage. A system has been developed to support this process. It takes the C program to be tested and instruments it with probes to provide feedback on the coverage achieved. The system creates an initial population of random data based on a description of the input data then performs an iterative search, which involves running this data and measuring its coverage (and hence, fitness). A sample of this population is selected (depending on the fitness value) to go forward to the new population and proportion of this new population is then subjected to mutation and crossover. The process is then applied again until a maximum level of fitness is reached by the test data. Jones et. al. [21] developed a GA for test-data generation for branch coverage. They use a control flow graph that represents one, two, and three iterations of each loop; because their representation unrolls each loop a specified number of times, their control flow graphs are acyclic. A program is instrumented so that as it executes with a test case, it records the branches it reaches and fitness of that test case. The fitness function uses the branch value, along with the value of the branch condition, to determine the fitness of the test case. The authors implemented the approach and preformed experiments with number of small programs. Sthamer [22] studied the use of GA as a test data generator for structural white box testing: branch, boundary, loop testing, and mutation testing. His example programs are small programs written in Ada including triangle classification, linear search, remainder calculation, and direct sort. Sthamer's fitness function is based on the predicates of the software under test. He observed that GAs show good results in searching the input domain for the required test sets. Weichselbaum [23] measured the coverage acquired by a test datum on the basis of the control flow graph. Weichselbaum concentrated on statement, branch, and condition testing. Pargas et. al. [24] presented GenerateData, an algorithm for automatic test-data generation for a given program. GenerateData uses a genetic algorithm, directed by the control-dependence graph of the program, to search for test data to satisfy test requirements. The test-data generation technique was implemented in a tool called TGen in which parallel processing was used to improve the performance of the search. The prototype, TGen, is implemented for statement and branch coverage. The algorithm evaluates test data by executing the program with the test data as input, and recording the predicates in the program that execute with that test data. This list of predicates is compared with the set of predicates found on the control-dependence predicate paths for the node representing the current test requirement that is the target of the search. A test data's fitness evaluation depends on the number of predicates that it has in common with the predicates on a control-dependence predicate path of the target: a solution that covers the greatest number of predicates is given the highest fitness evaluation. To experiment with TGen, a random test-data generator, called Random, was also implemented. Both TGen and Random were used to experiment with the generation of test data for statement and branch coverage of six programs. This approach clearly outperformed the random method for three of the six test programs, for the other three programs both methods find the optimal solution in the initial population. The work of Tracey [25] deals with automatic test-data generation for testing safety-critical systems. He uses simulated annealing and genetic algorithms, but also random search and hill climbing as the optimization methods. Bueno and Jino presented in [26-27] a new technique for path oriented test data generation for programs and identification of a path's likely unfeasibility in structural software testing. They propose that monitoring the progress of the GA search could identify an infeasible path. Their approach combines earlier works by other authors and introduces a new fitness function using control and data flow information to guide the search. They use the so-called ''path similarity metric'' as their fitness function. The authors present a new technique for choosing the initial search point using "past information" to improve the performance of test-data generation. Results are presented from an empirical evaluation done to assess the cost and the effectiveness of test-data generation using the proposed technique. Infinite loops are avoided by making the program execution halt if the number of traversed nodes is greater than a specified limit. Results with their six small test programs were promising. In the work of Wegener et. al. [28-29], development a test environment to support all common control-flow and data-flow oriented test methods. Also, several new structure-oriented fitness functions were introduced for most coverage types but their tool environment applied for automatic generation of test data for statement and branch testing. They introduced the idea of an approximation level, indicates how many branching condition nodes still require execution in the desired way to achieve the required partial aim. Lin and Yeh [30] have also studied automatic test-data generation by a GA for a chosen subpath. Their method uses a so-called ''normalized extended Hamming distance'' to guide the optimization process and to test the optimality of the candidate solutions. This fitness function, called SIMILARITY, defines how similar the traversed path is to the target path, is used to choose the surviving test cases. Optimality here means that the test case (i.e. a particular input) forces the program to follow the given path of program statements when executed. They claim that a GA is able to significantly reduce the time required for automatic path testing. Michael et. al [31] discussed the use of GAs for automatic software test-data generation. His work describes the implementation of GAs based system (GADGET), which attempted to generate test cases that satisfy condition-decision coverage criterion. This system (GADGET) was designed to work on programs written in C and C++. But this system is limited to programs whose inputs are scalar types. It can't intelligently handle Boolean variables or enumerated types. Michael et al. examined the effectiveness of this approach on a number of programs one of, which is significantly larger than those for which results have previously been reported in the literature. Also they examine the effect of program complexity on the test-data generation problem by executing this system on a number of synthetic programs that have varying complexities. Berndt et al. [32] distinguishes between absolute and relative fitness functions, that is used to organize past research and characterize this project's reliance on a relative or changing fitness function. In particular, the genetic algorithm includes a fossil record that records past organisms, allowing any current fitness calculations to be influenced by past generations. Three factors are developed for the fitness function: novelty, proximity, and severity. The interplay of these factors produces fairly complex search behaviors in the context of an example triangle program used in past software testing research. Lastly, several techniques for fossil record visualization are developed and used to analyze different fitness function weights and resulting search behaviors.

IV. Genetic-Based Test-Data generation techniques: A Comparison

This section present a comparison among the genetic algorithms based test-data generation techniques through many dimensions such as coverage criterion, fitness function, chromosome representation, the base of initial population selection, type and rate of crossover and mutation operators, population size, and the selection principle of the survival individuals. As shown in Table 1 for the first dimension, coverage criterion, Xanthakis, Pei, Watkins, Lin, and Bueno techniques are employed to generate test data for a selected set of paths of the program; each technique takes one path (not yet covered) at a time in the given sequence. Whereas, in the work of Xanthakis the genetic algorithm is used to find input data that satisfies all branch predicates of a chosen path. Pei's approach selects the set of paths of the program manually and selects the path limited to 2 loops, but Watkins attempts to obtain full path coverage for programs of a simple nature and containing no loops. Bueno's technique can be applied to the generation of test data for sub-paths from the entry node to some goal node different from the exit node.

TABLE I. COMPARISON ACCORDING TO COVERAGE CRITERION AND FITNESS FUNCTION

	Coverage Criterion	Fitness Function
Xanthakis	Path	The branch distance values.
Pei	Path (limited to 2 loops)	Fitness = C-[10*n+5*n(n-1)/2]
Watkins	Path	The function penalizes
Roper	Branch	Percentage of coverage achieved
Jones	Branch (with 0,1,2, and 3 loops)	Hamming distance or reciprocal
Pargas	Statement and branch	Common predicates
Lin	Path	Similarity
Michael	Branch (Condition-decision)	Predicate function
Bueno	Path	FT = NC - EP / MEP

Roper, Jones, Pargas, and Michael techniques attempt to achieve a desired level of branch coverage. Whereas, Jones technique attempts to ensure that all branches in the software were exercised but the loops are controlling to zero, one, two, and three loops. Pargas technique uses Control Dependence Graph thus the paths are acyclic and Michael's technique uses condition-decision coverage.

For the second dimension, fitness function, Xanthakis's fitness function is the sum of the branch predicates on the path, where a branch predicate has the form: E1 op E2 where E1 and E2 are arithmetic expressions and op is one of $\{<, \le, >, \ge, =, \ne\}$. Then, this branch predicate can be transformed to the equivalent function as shown in Table 2. Pei uses a most simple fitness function $= C - [10 \cdot N + 5 \cdot N(N-1)/2]$, where C is a big number, and N is a matching number between practical sub-paths and ideal required sub-paths. The third term is a scaling factor. Watkins's fitness function penalizes individuals that follow already covered paths, by assigning a value that is the inverse of the number of times the path has already been executed during the search. Roper's fitness function is the coverage of the program which achieved, i.e., the number of covered branches to the total number of branches. Jones considers two fitness functions: the Hamming distance function and a simple reciprocal of the difference between two predicate values. The former may be applied in general, while the latter applies only to predicates in which numerical values are compared. Pargas's fitness function is the number of predicates that it has in common with the predicates on a control-dependence predicate path of the target. Lin's technique uses a so-called ''normalized extended Hamming distance'' to guide the optimization process and to test the optimality of the candidate solutions. This fitness function, called SIMILARITY. Michael uses the fitness function shown in Table 3. If the program's execution fails to reach the desired location then the fitness function takes its worst possible value.

TABLE II. THE BRANCH FUNCTION

Branch Predicate	Branch function		When
E1 > E2	F =	E1 - E2	E1 - E2 > 0
E1 $\ge$ E2		0	E1 - E2 < 0
E1 < E2	F =	E2 – E1	E2 – E1 > 0
E1 $\le$ E2		0	E2 – E1 < 0
E1 = E2	F =	Abs(E1 - E2)	Abs(E1 - E2) > 0
E1 $\ne$ E2		0	Abs(E1 - E2) < 0

TABLE III. THE MICHAEL'S FITNESS FUNCTION

Decision type	Example	Fitness function
Inequality	If(c>= d)	d-c, if d $\ge$ c, 0, otherwise
Equality	If(c= = d)	$\lvert d - c \rvert$
Boolean value	If (c)	1000, if c=false 0, otherwise

Finally Bueno's technique uses the fitness function $FT = NC - EP/MEP$, where NC path similarity, EP absolute value of the path predicate (branch) function, and MEP is the maximum predicate function value among the candidate. Table 4 shows the comparison between some of the surveyed techniques according to genetic algorithm dimensions. From this comparison we note that these techniques use one of four chromosome representation binary string, gray code, character string, and list of input data. All techniques select the initial population randomly except in Bueno's technique the input data sets whose executed paths are similar to the desired one are recovered to be the initial population. All techniques use the single point crossover operator with rate from 0.60 to 0.90 except Jones uses uniform crossover with rate equal to 0.50. Also, Jones uses the reciprocal and weighted mutation but the others use the simple mutation with rate from 0.001 to 0.10. Each technique has a different population size. Many approaches are used to select the survival individuals such as high fitness, high average, high fitness in a selected subpopulation, hybrid between random and high fitness.

V. THE LIMITATIONS OF THESE TECHNIQUES

The new features of GAs make the existing test-data generation techniques based on them capable to find the nearly global optimum. However, these techniques have the following limitations:

A. Using Control Flow Coverage

Previous researches concentrated only on using control flow coverage criteria (e.g. statement, branch, path and condition-decision) and developing an appropriate fitness function definition for each criterion. But really, no one uses a data flow coverage criterion and there is no experiments do to discover the problems with these type of coverage criteria. However, higher levels of coverage may further discriminate among different test-data generation techniques. It would be interesting to apply these techniques to multiple condition coverage as well as data-flow coverage criteria.

TABLE IV. COMPARISON ACCORDING TO GENETIC ALGORITHM PARAMETERS

	Chromosome Representation	Initial Population Selection	Crossover operator Type	Crossover operator Rate	Mutation operator Type	Mutation operator Rate	Population Size	Selection
Pei	Binary string	Randomly	Single point	0.60-0.70	Simple Mutation	0.001	Program's size or paths	High fitness
Roper	Character string	Randomly	Single point	Input by the user	Simple Mutation	Input by the user	Input by the user	High fitness or average
Jones	Binary-plus-sign & gray code	Randomly	Uniform crossover	0.5	Reciprocal &Weighted	Reciprocal &five least	45	Hybrid
Pargas	List of input data	Randomly	Single point	0.90	Simple Mutation	0.10	100	High fitness
Michael	Binary string	Randomly	Single point	0.50	Simple Mutation	0.001	24 or 100	High fitness
Bueno	Binary string	Data of similar path	Single point	0.80	Simple Mutation	0.03	Around 80	f_i / f_{avg}

B. *Using Simple Types of Genetic Operators*

In spite of, there are many types of genetic operators and overlooking of, genetic operators which can specialize for test-data generation. Previous techniques concentrated only on using simple types of genetic operators (crossover and mutation) which sometimes destroy the input's data types (e.g. a simple mutation can change a string into an unprintable character).

C. *Not Considered Some Data Types and Multiple Procedures*

A structure is a collection of one or more variables possibly of different types grouped together under a single name for convenient handling. A pointer is a variable that contains the address of another variable actually it represents two variables: pointer itself and the variable pointed at. The problem here is how to solve the test-data generation problem using a GA in the presence of pointers to structures. The main problem for the search procedure using GA is to look for a suitable representation or coding for the structures and to design some recombination operation corresponding to the new representation.

D. *Manually Selecting the Set of Paths to Be Covered*

Path selection is the use of heuristics to choose an execution path that simplifies test-data generation. Although path selection is not vital in most of previous test-data generation techniques, it may still be the case that some execution paths are better than others for satisfying a particular test requirement. If static or dynamic analysis can provide clues about which paths are best, it will not be difficult to bias a genetic search algorithm toward solutions using those paths.

E. *Randomly Selecting the Initial Population*

All previous test-data generation techniques select the initial population randomly. These techniques can be improved by basing the initial population on a partial solution (e.g. a set of functional tests) rather than a random population, and use the system to fill in the gaps which the functional tests have missed. Also, the initial population can construct by a simple technique. These improvements can drive the technique to obtain the optimal solution quickly.

F. *Using Solid Fitness Functions*

When genetic search generates an input that fails to satisfy a particular test requirement that it is currently trying to satisfy, that input is simply given a low fitness value. However, there is at least one input that reaches the test requirement because of the way the algorithm is defined. If the technique assigns higher fitnesses to inputs that are closer to satisfy the test requirement, it might be possible to breed more inputs that actually reach it.

1) Control Dependences Related Problems for Fitness Functions

The fitness function that is used to optimize a test datum to execute a certain target node, as described in [24], takes control dependencies into account. This fitness function faces a problem to find an input to traverse a target node within loops [33]. This results in poor search performance. Jones et al. [21] avoid this problem by unrolling the loop in the control flow graph for the fitness evaluation only. The approach taken by Baresel et al. [33] is add dependencies of one loop iteration to the fitness function. Whilst monitoring the execution of the test object, one can observe this information on all iterations and calculate fitness from it. In order to circumvent this problem, Tracey [25] examines the branch distance during each iteration of the loop and uses the minimum branch distance obtained for the purposes of computing the final fitness value. A further problem is the assignment of approximation levels for some classes of program with unstructured control flow. Baresel et al. [33] present an example for this problem. Two plausible solutions to this problem include optimistic and pessimistic approximation level allocation strategies. In an optimistic strategy, a control dependent branching node is allocated its approximation level on the basis of the shortest control dependent path from itself to the target node. In a pessimistic strategy, a branching node is allocated its approximation level on the basis of the longest control dependent path to the target node. Both optimistic and pessimistic schemes were put to the test in initial experiments by Baresel et al. [33]. Whilst they show that the different schemes have different effects on the progress of the search, they were unable to conclude from the experiments which strategy works best in general. Thus, this problem is still open to question.

2) Branch-Distance-Related Problems for Fitness Functions

Although global search techniques are more robust than local searches in fitness function landscapes containing local optima and plateaus, they will still struggle in hostile search landscapes containing large plateaus or several local optima. In particular, plateaus can be induced on the search space through the use of "flag" variables in branch predicates. When flag variables are involved in branch predicates, the resulting fitness function landscape consists of two plateaus - one for the true value and one for the false value. In such situations, the evolutionary search performs no better than a random search. Bottaci [34] proposes a solution for a special case of flag problems, where the value of the flag is determined by a predicate. In this work it is suggested that the predicate used for the distance calculation is substituted by the predicate used in assigning the flag value, which provides more guidance to the required test data. However, flags are more commonly assigned constant true or false values. Harman et al. [35] suggest the use of a program transformation to remove flag variables from branch predicates, replacing them with the expression that led to their determination. In the transformed version of the program, the branch predicate is flag-free, and consequently plateaux induced by the flag are also removed. Note that although the flag is removed from the branch predicate, it otherwise remains present in the program, in case it has a future purpose in a later statement. A disadvantage of the approach is that it can not yet transform programs where flags are involved in loops. The approach of Baresel and Sthamer [36] is to identify a sequence of nodes to be executed prior to the branch predicate containing the flag. The sequence of nodes to be executed is performed via data- flow analysis of the flags involved.

A further problem can occur with nested branch predicates [33]. Once input data is found for one or more of the predicates, the chances of finding input data that also fits subsequent predicates decreases. This is because a solution for subsequent conditions must be found without violating any of the earlier conditions. This leads to poor search performance. Ideally, all of the conditions should be evaluated at once. Such a situation could be established through the use of data dependency analysis [33]. A similar problem occurs with the use of short circuit evaluation of atomic conditions with branch predicates using operators such as && and || in C. In such situations the evaluation of the overall predicate breaks off early if the end result has already been determined. Therefore, during the process of searching for test data, the individual conditions have to be attempted one after the other. Again, it would be preferable to evaluate all of the conditions at once. In this situation, care needs to be taken when side effects appear in any of the conditions. A solution here might be to apply a side-effect removal program transformation [37-38] first. Alternatively, variables values could be saved into temporary variables inserted immediately before the branching statement, and restored after the side-effect if the condition would not normally have been evaluated.

VI. CONCLUSIONS AND FUTURE WORK

The new features of GAs make the test-data generation process easily and find the nearly global optimum. We have described in section 6 some limitation of the test-data generation techniques based on genetic algorithms such as they concentrated only on: using control flow coverage, using simple types of genetic operators, not considered test-data generation in the presence of pointers, dynamic data structures, and multiple procedures, manually selected the set of paths to be covered, randomly selected the initial population, and using solid fitness functions. Furthermore, there are other problems as flag and enumeration variables and unstructured control flow. Additional researches are required to overcome these problems.

REFERENCES

[1] Frankl P G, Weiss S N. An experimental comparison of the effectiveness of branch testing and data flow testing. IEEE Transactions on Software Engineering, 19(8), 774-787, 1993.

[2] Mills H D, Dyer M D, Linger R C. Cleanroom Software Engineering. IEEE Software 4(5), 19-25, September 1987.

[3] Voas J M, Morell L, Miller K W. Predicting where faults can hide from testing. IEEE, 8(2), 41-48, March 1991.

[4] Thévenod-Fosse P, Waeselynck H. STATEMATE: Applied to Statistical Software Testing. ACM SIGSOFT. Proceedings of the 1993 International Symposium on Software Testing and Analysis, Software Engineering Notes 23(2), pp. 78-81, June 1993.

[5] Boyer R S, Elspas B, Levitt K N. SELECT - a formal system for testing and debugging programs by symbolic execution. Proceedings of the International Conference on Reliable software, 234-245 (1975).

[6] Clarke L A. A system to generate test data and symbolically execute programs. IEEE Transactions on Software Engineering, 2(3), 215-222, 1976.

[7] King J C. Symbolic execution and program testing. Communications of the ACM, 19 (7), 385-394, 1976.

[8] Howden W E. Symbolic testing and the DISSECT Symbolic evaluation system. IEEE Transactions on Software Engineering, 3(4), 266-278, 1977.

[9] Hedley D, Hennell M A. The causes and effects of infeasible paths in computer programs. Proceedings of Eighth International Conference on Software Engineering, IEEE Computer Society, 259-266, 1985.

[10] Lindquist T E, Jenkins J R. Test-case generation with IOGen. IEEE Software, 5 (1), 72-79 (1988).

[11] Girgis M R. An experimental evaluation of a symbolic execution system. Software. Engineering Journal, 7(4), 285-290, 1992.

[12] Girgis M R. Using symbolic execution and data flow criteria to aid test data selection. The Journal of Software Testing, Verification and Reliability, 3(2), 101-112, 1993.

[13] Korel B. Automated Software Test Data Generation. IEEE Transactions on Software Engineering, 16(8): 870-879, August 1990.

[14] Ferguson R, Korel B. The Chaining Approach for Software Test Data Generation. ACM TOSEM, vol. 5, no. 1, pages 63-86, January 1996.

[15] McMinn P. Search-based Software Test Data Generation: A Survey. Journal of Software Testing Verification and Reliability, vol.14, no.2, pp.105-156, June 2004.

[16] Mantere T, Alander J T. Evolutionary Software Engineering, A Review. Journal of Applied Soft Computing, vol.5, pp.315-331, 2005.

[17] Xanthakis S, Ellis C, Skourlas C, Le Gall A, Kastiskas S, Karapoulios K. Application of genetic algorithms to software testing (Application des algorithms génétiques au test des logiciels). In 5th International Conference on Software Engineering and its Applications, pages 625-636, Toulouse, France, 1992.

[18] Pei M,. Goodman E D, Gao Z, Zhong K. Automated Software Test Data Generation Using A Genetic Algorithm. Technical Report GARAGe of Michigan State University June 1994.

[19] Watkins A. A tool for the automatic generation of test data using genetic algorithms. In Proceedings of the fourth Software Quality Conference, Dundee, Great Britain, pp. 300-309, 1995.

[20] Roper M, Maclean I, Brooks A, Miller J, Wood M. Genetic Algorithms and the Automatic Generation of Test Data. Technical report RR/95/195[EFoCS-19-95], Department of Computer Science, University of Strathclyde, 1995.

[21] Jones B F, Sthamer H H, Eyres D E. Automatic Structural Testing Using Genetic Algorithms. Software Engineering Research Journal, pp. 299-306, September 1996.

[22] Sthamer H. H., the automatic generation of software test data using genetic algorithms, Ph.D. Thesis, University of Glamorgan, Pontyprid, Wales, Great Britain, 1996.

[23] Weichselbaum R., software test automation by means of genetic algorithms, Proceedings of the Sixth International conference on Software testing, Analysis and review (EuroSTAR 98), Munich Germany (1998).

[24] Pargas R P, Harrold M J, Peck R R. Test Data Generation Using Genetic Algorithms. Journal of Software Testing, Verifications, and Reliability, vol. 9, pp. 263-282, September 1999.

[25] Tracey N., A Search-Based Automated Test Data Generation Framework for Safety Critical Software. Ph. D. thesis, University of York, 2000.

[26] Bueno P M S, Jino M. Identification of Potentially Infeasible Program Paths by Monitoring the Search for Test Data. Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00).

[27] Bueno P M S, Jino M. Automatic Test Data Generation for Program Paths Using Genetic Algorithms. International Journal of Software Engineering and Knowledge Engineering, vol. 12, no. 6, pp. 691-709, 2002.

[28] Wegener J., Baresel A., Sthamer H.. Evolutionary test environment for automatic structural testing. Journal of Information and Software Technology, vol. 43, pp. 841-854, 2001.

[29] J. Wegener, K. Buhr, and Pohlheim H.. Automatic test data generation for structural testing of embedded software systems by evolutionary testing. Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002), pp. 1233-1240, New York, UAS, 2002.

[30] Lin J, Yeh P. Automatic Test Data Generation for Path Testing Using GAs. Journal of Information Science vol. 131, pp. 47-64, 2001.

[31] Michael C C, McGraw G E, Schatz M A. Generating Software Test Data by Evolution. IEEE Transactions on Software Engineering, vol.27, no.12, pp. 1085-1110, December 2001.

[32] Berndt D, Fisher J, Johnson L , Pinglikar J, Watkins A. Breeding Software Test Cases with Genetic Algorithms, Proceedings of 36th Hawaii International Conference On System Sciences 2003.

[33] Baresel A, Sthamer H, Schmidt M. Fitness Function Design to Improve Evolutionary Structural Testing. In proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002), pages 1329-1336, New York, USA, 2002.

[34] Bottaci L. Instrumenting programs with flag variables for test data search by genetic algorithm. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002), pages 1337-1342, New York, USA, 2002. Morgan Kaufmann.

[35] Harman M, Hu L, Hierons R, Baresel A, Sthamer H. Improving evolutionary testing by flag removal. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002), pages 1359-1366, New York, USA, 2002. Morgan Kaufmann.

[36] Baresel A, Sthamer H. Evolutionary testing of flag conditions. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003), Lecture Notes in Computer Science vol. 2724, pages 2442-2454, Chicago, USA, 2003. Springer-Verlag.

[37] Harman M, Hu L, Zhang X, and Munro M. Side-effect removal transformation. In Proceedings of the 9th IEEE International Workshop on Program Comprehension (IWPC2001), pages 310-319, Toronto, Canada, 2001. IEEE Computer Society Press.

[38] Harman M, Hu L, Zhang X, Munro M, Dolado J J, Otero M C, Wegener J. A post-placement side-effect removal algorithm. In Proceedings of the 18th IEEE International Conference on Software Maintenance (ICSM 2002), pp 2-11, Montreal, Canada, 2002.

[39] Baresel A, Pohlheim H, and Sadeghipour S. Structural and functional sequence test of dynamic and state-based software with evolutionary algorithms. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003), Lecture Notes in Computer Science vol. 2724, pages 2428 - 2441, Chicago, USA, 2003. Springer-Verlag.

[40] McMinn P, Holcombe M. The state problem for evolutionary testing. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003), Lecture Notes in Computer Science vol. 2724, pages 2488-2497, Chicago, USA, 2003. Springer-Verlag.