Volume 4, Issue 2, February 2014 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Automated Software Test Data Generation for Data Flow Dependencies using Genetic Algorithm

Sapna Varshney[*], **Monica Mehrotra**
Department of Computer Science
Jamia Millia Islamia, India

Abstract— Software testing is one of the most labor-intensive and expensive phase of the software development life cycle. Software testing includes test case generation and test suite optimization that has a strong impact on the effectiveness and efficiency of software testing. Over the past few decades, there has been active research to automate the process of test case generation but the attempts have been constrained by the size and the complexity of software. The use of metaheuristic global search techniques for software test data generation has been the focus of researchers in recent years. Many new techniques and hybrid methods have also been proposed to tackle the problem more effectively. This study proposes a novel approach based on genetic algorithm to generate test data for a program. The performance of the proposed approach is evaluated based on data flow dependencies of a program by comparing it with random testing. Based on the experimental results on a number of C programs, it is shown that the proposed approach outperforms random testing in test data generation and optimization.

Keywords—Search Based Software Testing, Automated Test Data Generation, Metaheuristic Search Algorithms, Evolutionary Algorithms, Genetic Algorithms, Data Flow Dependencies

I. INTRODUCTION

Software testing is the process of executing software with the aim of detecting as many defects as possible so as to assess the quality of the developed software. Software testing increases programmers' as well as the users' confidence in the correctness and reliability of the software. Exhaustive testing is not possible as the input search space grows exponentially with the number of input variables [23]. The goal of software testing is to generate an optimal test suite (set of test cases) that reveals as many errors as possible according to a test adequacy criterion.A test adequacy criterion distinguishes good test cases from bad ones and determines whether the testing process is finished. There have been constant attempts to reduce the efforts and time required for software testing by automating the process of software test data generation. In the early period of software testing automation, most of the test data generators were based on the gradient descent and local metaheuristic search (MHS) algorithms such as Tabu Search (TS) and Hill Climbing (HC). However, these algorithms are inefficient and time-consuming, and could return a local optimal solution in the input search space [13] [29]. Other global MHS algorithms, such as Simulated Annealing (SA) [24], have been employed for test data generation; however, there are still chances of obtaining a local optimal solution. In the past two decades, evolutionary search based algorithms, such as Genetic Algorithm (GA) [2] [3] [13] [18] [23], have been widely employed for test data generation as a better alternative. Each of these search based algorithms is strongly dependent on the domain of the problem under consideration because they use heuristics or the knowledge related to the problem domain. The widespread application of MHS algorithms for test data generation problem is because of the fact that it can be formulated as an optimization problem. The approach has come to be known as Search Based Software Testing (SBST, the term originally coined by Harman and Jones in 2001) and includes Evolutionary Testing as a sub-field.
This study proposes a new approach based on Genetic Algorithm to automatically generate test data using data flow dependencies of a program. The performance of the proposed approach is compared with random testing due to its simplicity, efficiency and efficacy in terms of achieving coverage and the number of test cases generated.
The rest of the document is organized as follows: Section 2 provides an overview of software testing and automated software test data generation process. Section 3 provides a brief description of genetic algorithm. Section 4 describes the proposed approach. Section 5gives the experimental results and section 6 gives the conclusion. The various repositories and search engines that have been referenced for literary articles and papers for this study are ACM Digital Library, IEEE Explore, Springer Verlag, Science Direct, Google Scholar and CiteSeer.

II. SOFTWARE TESTING

Software testing [7] only reveals the presence of errors in a program but never guarantees their absence. It increases the programmers' as well as the users' confidence in the correctness and reliability of the software. Software testing techniques are classified into two categories – static testing and dynamic testing. In *static software testing*, specification documents, design documents and source code of the software under test (SUT) are examined. Static analysis methods (desk checking, code reviews) are highly dependent on the reviewers' experience and ability. In *dynamic software testing,* the output is observed by executing the SUT on input test data. The SUT is tested for its functionality (*functional* or *specifications-based* or *black box testing)* or for its structure (*structural* or *program-based* or *white box testing).*

In *black-box testing,* test data is generated from the specifications of the SUT. The black-box testing techniques include equivalence partitioning, boundary-value analysis, cause-effect graphing etc.

In *white-box testing,* the internal structure of the SUT is examined by executing the code such that every statement in the SUT should be executed at least once (*Statement Testing*); or every possible outcome of all decisions/predicates in the SUT should be exercised at least once (*Branch Testing*); or every possible path in the SUT should be exercised at least once (*Path Testing*). Branch testing includes statement coverage and is a stronger criterion than statement coverage. Path testing includes statement coverage and branch coverage and thus is a stronger criterion than statement and branch testing. In *data-flow testing,* the focus is on the definition and use of variables within a program by utilising the concept of a program graph.

A. Data Flow Testing

Data-flow testing is important because it augments control-flow testing criteria and concentrates on how a variable is defined and used, which could lead to more efficient and targeted test suites. For each variable, the definition occurrences and the use occurrences are identified. A definition occurrence of a variable is where a value is associated with the variable. A use occurrence of a variable is where the value of the variable is referred. Each use occurrence is further classified as a computational use (c-use) or a predicate use (p-use). Test data for data flow testing should cause the traversal of sub-paths from a variable definition to either some or all of the p-uses, c-uses, or their combination. However, empirical evidences show that the all-uses criterion is the most effective criterion compared to the other data flow criteria. It requires the traversal of at least one sub-path (def-clear path) from each variable definition to every p-use and every c-use of that definition. A def-clear path is a path from definition node to use node such that the variable is not defined again at any of the intermediate nodes.

Control Flow Graph (CFG): A CFG is a directed graph that represents the flow of control through a program. It describes the sequence in which the statements of a program are executed. Each node represents a basic block i.e. a sequence of consecutive statements that executes without any halt or branching except at the end. Each edge represents the flow of control from one basic block to another. All edges are labelled with a condition or a branch predicate. If a node has more than one outgoing edge the node represents a condition and the edge represents branch. A CFG has two special nodes: the entry node, through which control enters into the flow graph, and the exit node, through which all control flow leaves.

Data Dependency Graph (DDG): A DDG, also known as a data flow graph, represents data dependencies between the statements of a program. Nodes in a data flow graph represent statements where memory references are made i.e. variables are defined or used. Edges represent data dependencies between nodes. A data dependency is said to exist between statements S1 and S2 of a program, if S2 references a variable defined in S1 and there is a feasible run-time path from S1 to S2 on which the variable is not defined again; then (S1, S2) is a definition-use pair. A DDG can be generated from a CFG by using data dependency information.

This study will focus on structural testing as it is the most widely practiced form of testing and more specifically on data-flow testing as it has received little attention [28]. Moreover, data-flow testing could lead to more efficient and targeted test suites as it augments control flow testing criteria with the definition and usage of variables.

B. Automated Software Test Data Generation

Software testing has two main aspects: test data generation and application of a test adequacy criterion. Structural test data can be generated using static methods or dynamic methods. *Symbolic execution and evaluation* is a typical static tool for generating test data. In symbolic execution, expressions are assigned to program variables as a path is followed through the code structure to derive constraints in terms of the input variables [24]. However, symbolic execution suffers from many drawbacks such as input variable dependent loop conditions and array reference subscripts, module calls and pointers [4].

Dynamic structural test data generation techniques are based on the execution of the SUT to generate test data. These techniques can be classified as *random*, *structural or path-oriented*, *goal-oriented* and *data specifications* test data generation techniques [4] [27]. ***Random test data generators*** arbitrarily select test data from the input domain. A random test data generator can randomly create a large number of test data and is easy to implement; however, it may fail to find test data as the information about test requirements is not incorporated into the test data generation process. ***Path-oriented test data generators*** [6] [12] [13] generally use control flow information (by constructing the program's control flow graph) to identify a set of paths to be covered and generate the appropriate test cases for these paths. The test data generator will not work well for infeasible paths or paths that contain loops. ***Goal-oriented test data generators*** [4] [11] [16] [21] [25] [27] identify test cases covering a selected goal such as a statement or branch, irrespective of the path taken. ***Data specification generators*** derive test data from specifications (a black-box method).

Structural testing so far has been the main focus of search based techniques. Test data is generated according to a test adequacy criterion (encoded as a fitness function) that is used to guide the search. The *fitness function* captures a test objective that should be maximized or minimized. The search based approach is very generic, because different fitness functions can be defined to capture different test objectives, allowing the same overall search based optimization strategy to be applied to very different test data generation scenarios.

The measures that have been used to assess the effectiveness of a search based technique for structural testing are control flow coverage (statement coverage, branch coverage, path coverage), data flow coverage and N-wise coverage (for testing combinatorial designs) [28]. Control-flow based coverage criteria, branch coverage in particular, are the most often used effectiveness measures. As a result, this problem is now pretty well understood and there is a widely accepted standard way of calculating fitness values based on approximation level and branch distance on control flow graphs [23]. Data flow coverage criterion [2] [3] [18] [19] [21] has received relatively little attention. The measures that have been used to assess the cost of a search based technique for structural testing are the number of iterations, the cumulative number of all individuals in all iterations, the number of fitness evaluations performed to achieve the final solution, size of the optimal test suite and the time taken to generate the optimal test suite [28]. The number of iterations and the size of the optimal test suite are the most often used cost measures.

III. GENETIC ALGORITHM

Genetic Algorithm (GA) is a population based search algorithm that works on the principle of natural evolution (crossover and mutation) and selection leading to the survival of the fittest individuals. GA has been the most widely applied search technique in SBSE. GA creates and maintains a population of individuals represented by chromosomes. These chromosomes are typically encoded solutions to a problem. Each chromosome receives a measure of its fitness in the environment. The chromosomes then undergo a process of evolution according to rules of selection, mutation and reproduction. Reproduction selects individuals with high fitness values in the population, and through crossover and mutation of such individuals, a new population is derived in which individuals may be even better fitted to their environment. The process of crossover involves two chromosomes swapping chunks of data (genetic information). Mutation introduces slight changes into a small proportion of the population and is representative of an evolutionary step. The structure of a simple GA is given below in Figure 1. The algorithm will iterate until the population has evolved to form a solution to the problem, or until a maximum number of iterations have taken place (suggesting that a solution is not going to be found given the resources available).

```
1. Randomly generate or seed initial population P
2. Repeat
3.        Evaluate fitness of each individual in P
4.        Select parents from P according to selection
          mechanism
5.        Recombine parents to form new offspring
6.        Construct new population P’ from parents and
          offspring
7.        Mutate P’
8.        P = P’
9. Until Stopping Condition Reached
```

There are many variations of genetic algorithm, but the crucial ingredients are the way in which the fitness function guides the search and the recombination and the population based nature of the algorithm.

IV. RESEARCH METHODOLOGY

This paper presents an automatic test data generation technique based on GA that is guided by the data flow dependencies in the program (test adequacy criteria). The approach can be used for programs with/without loops and procedures. The proposed GA accepts as input an instrumented version of the program to be tested, the list of def-use associations to be covered, the number of input variables, and the domain and precision of each input variable. The algorithm produces a set of test cases, the set of def-use associations covered by each test case, and a list of uncovered def-use associations, if any.

A. *GA Setup*

1) *Chromosome Representation:* A chromosome is represented as a binary vector of length l determined by the number of input variables in a program and the domain range for each input variable. For example, if there are 3 variables in a program and value of each variable is represented by 7 bits, then the first 7 bits (MSBs) will map to a value in the domain range of first input variable, next 7 bits will map to a value in the domain range of second input variable and the last 7 bits (LSBs) will map to a value in the domain range of third input variable; total length of the chromosome being 21 bits.

2) *Initial Population:* Initial population of l-bit binary strings (chromosomes) is generated randomly. Size of the population, p, is determined experimentally. Each chromosome in the population is converted to k decimal numbers corresponding to the integer input variables x_1, $x_{2...}x_k$ according to the formula [18] given below:

$$\mathrm{x_i} = \mathrm{a_i} + int\left(x_i'.\frac{b_i - a_i}{2^{\mathrm{m_i}} - 1}\right)$$

, where [a_i, b_i] is the domain range of the input variable x_i, m_i is the number of bits used to represent the value of x_i in binary and x_i' represents the decimal value of the binary string corresponding to x_i.

3) *Fitness Function and Selection:* Each test case, represented as a chromosome, is evaluated by executing the program with it as an input, and recording the def-use paths in the program that are covered by it. The fitness value for each chromosome, v_i (i=1…p) is calculated as follows:

$$Fitness\ (v_i) = \frac{Number\ of\ new\ dcu\ paths + dpu\ paths\ covered}{Total\ number\ of\ dcu\ paths\ and\ dpu\ paths}$$

The fitness value is the only feedback from the problem for the GA. A chromosome is considered effective if its fitness value is greater than 0.
Random selection method [18] is used to select parents according to their fitness for generating the next generation of population, if the test adequacy criterion is not achieved.

4) *Genetic Operations - Crossover and Mutation:* Crossover and mutation operators are used to form a new population from the selected parents. Crossover operates at the individual level. A chromosome in the current population is selected for crossover according to a crossover probability p_c. One point crossover is used for experimental setup. Each pair of selected chromosomes is mated at a random position crosspos and is replaced by the pair of their offspring. The expected number of chromosomes that will undergo crossover operation is given by p_c x p. Mutation operation is applied after the crossover operation. In the experiments, mutation is applied in two ways: at an individual level and on a bit-by-bit basis. In the first case, a chromosome in the current population is selected for mutation according to a mutation probability p_m and one bit is flipped randomly for each input variable for the selected chromosome. The expected number of chromosomes that will undergo mutation operation is given by p_m x p. In the second case, for each chromosome in the current population, each bit is flipped with the pre-determined probability. The expected number of mutated bits is given by p_m x l x p.

The population evolves until the desired level of coverage is achieved i.e. when a set of individuals has traversed the entire def-use paths of program, if possible. The solution is this set.

B. Overall Algorithm

The proposed genetic algorithm accepts as input an instrumented version of the program to be tested, the list of def-use paths to be covered, the number of input variables, and the domain and precision of each input variable. It also accepts the GA parameters: population size, maximum number of generations, and probabilities of the crossover and mutation. The algorithm produces a set of test cases, the set of def-use paths covered by each test case, and the list of uncovered def-use paths, if any. Instrumentation of programs and generation of the program def-use paths is done manually.
The algorithm uses an integer vector to indicate whether a def-use path has been covered by some test case or not. In this vector, each element (initially zero) corresponds to a def-use path. Whenever a def-use path is covered, corresponding element is updated to 1. For each test case, the new def-use paths that are covered by it are recorded to calculate incremental coverage achieved and its fitness value. The algorithm keeps track of all the test cases that cover any new def-use path. These test cases are stored for later use and form the effective members of the current population. The effective members are used to select parents randomly for the next generation. If there are no effective members in the current population, then the entire current population is used to select parents randomly for the next generation. When the algorithm terminates, elements with value 0, if any, indicate def-use paths that have not been covered by any of the test case.

V. EXPERIMENTAL RESULTS

The proposed algorithm is applied to a number of classical C programs such as quadratic equation problem, triangle classification problem, date problem etc. A random test data generator is also implemented for comparison with the proposed approach. For the experiments,

- Input variables are of type integer, range 0-100 and 7 bits are used to represent each variable.
- Population size=5, 10, 20
- Fitness function, crossover operation and mutation operation are applied as explained in Section 4.
- Parent selection method: Random selection method
- Crossover probability=0.8
- Mutation probability=0.15

The approach is explained on triangle classification problem as given in Figure 2 below. Figure 3 gives the program flow graph for the example program. Table 1 provides definition and use nodes for each variable in the example program. Table 2 provides the list of dcu-paths and dpu-paths for the example program.

```
#include<stdio.h>
#include<conio.h>
1    1 void main()
2    1 {
3    1    int a, b, c, valid;
4    1    printf("\nEnter the value of three sides: ");
5    1    scanf("%d %d %d", &a, &b, &c);
6    1    valid=0;
7    2
     if((a>=0)&&(a<=100)&&(b>=0)&&(b<=100)&&(c>=0)&&(c<=100))
8    2    {
9    3              if(((a+b)>c)&&((c+a)>b)&&((b+c)>a))
10   3              {
11   4                        valid=1;
12   5              }
13   5    }
14   6    if (valid==1)
15   6    {
16   7              if ((a==b)&&(b==c))
17   8                   printf("\nEquilateral triangle.");
18   9              else if ((a==b)||(b==c)||(c==a))
19   10                  printf("\nIsosceles triangle.");
20   11             else
21   11                  printf(("\nScalene triangle.");
22   12   }
23   13   else
24   13   {
25   13             printf("\n Invalid input (out of range or not a triangle)").;
26   14   }
27   15 }
```

Figure2: Triangle program.

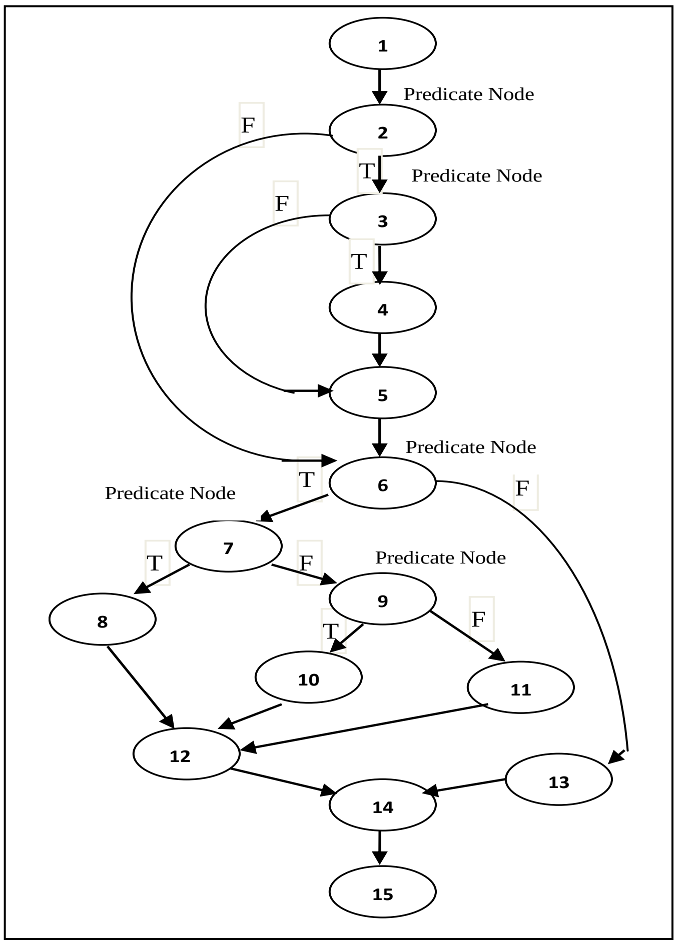

TABLE 1
Definition and Use Nodes for the Variables in the Example Program

Variable	**Def Node**	**c-use Node**	**p-use Edge**
A b c	1	None	2-3 2-6 3-4 3-5 7-8 7-9 9-10 9-11
Valid	1,4	None	6-7 6-13

TABLE 2
dcu-paths and dpu-paths for the Example Program

def-use Path No.	**def-use Path** (Terminates with -1 for c-use)
1	1-2-3
2	1-2-6
3	1-3-4
4	1-3-5
5	1-7-8
6	1-7-9
7	1-9-10
8	1-9-11
9	1-6-7
10	1-6-13
11	4-6-7
12	4-6-13

Table 3 below shows the results of applying the proposed approach and random testing to the set of chosen programs. For each program, infeasible paths, if any, were not considered while measuring coverage according to the def-use coverage criterion.

TABLE 3
Experimental results for various C programs

Program	**No. of Variables**	**No. of def-use Paths**	**Population Size**	**Testing Approach**	**Average No. of Generations***		**No. of Test Cases**
					>90% Coverage	**100% Coverage**	
Midval	**3**	**29**	**5**	**GA**	**2**	**3**	**4**
				Random	**2**	**3**	**4**
Quadratic	**5**	**20**	**10**	**GA**	**2**	**4**	**5**
				Random	**4**	**7**	**5**
Triangle	**4**	**11**	**10**	**GA**	**2**	**4**	**4**
				Random	**3**	**6**	**4**
GCD of 2 Numbers	**3**	**17**	**10**	**GA**	**2**	**4**	**3**
				Random	**3**	**5**	**3**
Avg. Marks of 3 Subjects	**4**	**14**	**10**	**GA**	**2**	**3**	**4**
				Random	**2**	**4**	**4**
Previous Date	**5**	**66**	**20**	**GA**	**19**	**37**	**10**
				Random	**35**	**95**	**13**
Power of a Number	**3**	**21**	**10**	**GA**	**2**	**3**	**2**
				Random	**2**	**3**	**2**
Prime Number	**2**	**13**	**5**	**GA**	**1**	**2**	**2**
				Random	**2**	**3**	**2**

* Number of runs of the algorithm=10

The proposed approach outperformed random testing in terms of the number of generations needed to achieve desired coverage and the size of the final test suite.

VI. CONCLUSION

Research in the field of search based test data generation for structural testing is now relatively mature with work on theoretical and empirical analysis of the problem domain and characteristics, predictive methods for test effort and search algorithms that are specially tailored to the structural test data generation problem [17]. The GA-based technique presented in this paper is guided by the data flow dependencies in the program to search for test data to fulfil the all-uses criterion. This is the main contribution of this paper. The approach can be used in test data generation for programs with/without loops and procedures. The proposed GA-based technique accepts as input an instrumented version of the program to be tested, the list of def-use paths to be covered, the number of input variables, and the domain and precision of each input variable. It also accepts the GA parameters: population size, maximum number of generations, probability of crossover operation and probability of mutation operation. The def-use paths that are infeasible (to be identified manually) are not considered while calculating coverage with respect to all-uses criterion. The algorithm produces a set of test cases (test suite), the set of def-use paths covered by the test suite, and a list of uncovered def-use paths, if any.

Experiments have been carried out to evaluate the effectiveness of the proposed GA compared to the random testing technique The results of these experiments showed that the proposed technique outperformed the random testing technique in most of the programs used in the experiment in terms of the number of generations required to achieve the same def-use coverage and the size of the resulting test suite. The experiments showed that the proposed technique achieved higher coverage percentage in fewer generations than the random testing technique.

FUTURE WORK

Future work for the proposed approach will focus on the following:

a) Design and application of a hybrid search technique, which combine the best aspects of existing search algorithms, for structural test data generation problem [11] [15] [17].
b) Multi-objective formulation of test data generation problem, focussing on Pareto optimal optimization techniques [16] [17].
c) Application of the other highly adaptive search techniques such as Particle Swarm Optimization (PSO) and Ant Colony Optimization (ACO) techniques that have yet not been widely applied and studied in search based software engineering [17].

REFERENCES

[1] A. Bertolino, “Software Testing Research: Achievements, Challenges, Dreams”, in *Proc. of the International Conference on Future of Software Engineering,* 2007.

[2] A. S. Andreou, K. A. Economides, and A. A. Sofokleous, “An automatic software test-generation scheme based on data flow criteria and genetic algorithms”, in *Proc. of the IEEE Seventh International Conference on Computer and Information Technology*, pp. 867-872, 2007.

[3] A. S. Ghiduk, M. J. Harrold, and M. R. Girgis, “Using Genetic Algorithms to Aid Test-Data Generation for Data-Flow Coverage”, in *Proc. of the IEEE 14th Asia-Pacific Software Engineering Conference*, pp. 41-48, 2007.

[4] B. Korel, “Automated Software Test Data Generation”, *IEEE Transactions on Software Engineering,* vol. 16(8), pp. 870-879, 1990.

[5] C. C. Michael, G. E. McGraw, and M. A. Schatz, “Generating Software Test Data by Evolution”, *IEEE Transactions on Software Engineering,* vol. 27(12), pp. 1085-1110, Dec. 2001.

[6] D. Gong and X. Yao, “Automatic detection of infeasible paths in software testing”, *IET Software,* vol. 4(5), pp. 361-370, 2010.

[7] G. J. Myers, *The Art of Software Testing,* New Jersey: Wiley, 2004.

[8] J. H Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, “Mutation Analysis for Assessing and Comparing Testing Coverage Criteria”, Carleton University, TR SCE-06-02, Mar. 2006.

[9] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection,* Cambridge: MIT Press, 1992.

[10] K. Lakhotia, “Search-Based Testing”, Doctoral Thesis, Department of Computer Science, King’s College, London, Oct. 2009.

[11] K. Lakhotia, P. McMinn, and M. Harman, “Automated Test Data Generation for Coverage: Haven’t We Solved This Problem Yet?”, in *Proc. of the 4thTesting Academia and Industry Conference - Practice and Research Techniques (TAIC PART 09),* UK, 2009.

[12] K. Li, Z. Zhang, and J. Kou, “Breeding Software Test Data with Genetic-Particle Swarm Mixed Algorithm”, *Journal of Computers*, vol. 5(2), pp. 258-265, Feb. 2010.

[13] M. A. Ahmed, and I. Hermadi, “GA-based multiple paths test data generator”, *Elsevier Computers and Operations Research,* vol. 35, pp. 3107-3124, 2007.

[14] M. Harman, “The Current State and Future of Search Based Software Engineering”, in *Proc. of the 29th International Conference on Software Engineering, Minneapolis, USA,* 2007.

[15] M. Harman, and P. McMinn, “A Theoretical and Empirical Study of Search Based Testing: Local, Global and Hybrid Search”, *IEEE Transactions on Software Engineering,* vol. 36(2), pp. 226-247, 2010.

[16] M. Harman, K. Lakhotia, and P. McMinn, "A Multi-Objective Approach To Search-Based Test Data Generation", *ACM GECCO,* pp. 1098-1105, 2007.

[17] M. Harman, S. A. Mansouri, and Y. Zhang, "Search Based Software Engineering: A Comprehensive Analysis and Review of Trends Techniques and Applications", TR-09-03, Apr. 2009.

[18] M. R. Girgis, "Automatic Test Data Generation for Data Flow Testing Using a Genetic Algorithm", *Journal of Universal computer Science,* vol. 11(6), pp. 898-915, 2005.

[19] N. Nayak, and D. P. Mohapatra, "Automatic Test Data Generation for Data Flow Testing Using Particle Swarm Optimization", *Springer-Verlag Berlin Heidelberg,* pp. 1-12, 2010.

[20] P McMinn, M. Harman, Y. Hassoun, K. Lakhotia, and J. Wegener, "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,* vol. 38(2), pp. 453–477, 2012.

[21] P. G. Frankl, and S. N. Weiss, "An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing", *IEEE Transactions on Software Engineering,* vol. 19(8), pp. 774-787, Aug. 1993.

[22] P. McMinn, "An Identification of Program Factors that Impact Crossover Performance in Evolutionary Test Input Generation for the Branch Coverage of C Programs", *Information and Software Technology,* vol. 55(1), pp. 153–172, Jan. 2013.

[23] P. McMinn, "Search-Based Software Test Data Generation: A Survey", *Journal of Software Testing, Verification and Reliability,* vol. 14(2), pp. 105-156, June 2004.

[24] R. A. DeMillo, and A. J. Ofutt, "Constraint-based automatic test data generation", *IEEE Transactions on Software Engineering,* vol. 17(9), pp. 900-910, Sep. 1991.

[25] R. Ferguson, and B. Korel, "The chaining approach for software test data generation", *ACM Transactions on Software Engineering and Methodology,* vol. 5(1), pp. 63-86, 1996.

[26] R. Malhotra, and M. Garg, "An Adequacy Based Test Data Generation Technique Using Genetic Algorithms", *Journal of Information Processing Systems,* vol. 7(2), pp. 363-384, June 2011.

[27] R. P. Pargas, M. J. Harrold, and R. Peck, "Test-Data Generation Using Genetic Algorithms", *Journal of Software Testing, Verification and Reliability,* vol. 9(4), pp. 263-282, 1999.

[28] S. Ali, L. C. Briand, H. Hemmati, and R. K. P Walawege, "A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation", *IEEE Transactions on Software Engineering,* vol. 36(6), pp. 742-762, Nov./Dec. 2010.

[29] X. S. Yang, *Engineering Optimization: An Introduction with Metaheuristic Applications,* New Jersey: Wiley, 2010.