Automatic Test Data Generation for Data Flow Testing Using Particle Swarm Optimization

Narmada Nayak* and Durga Prasad Mohapatra

Department of Computer Science and Engineering
National Institute of Technology Rourkela,
Orissa-769008, India
nayak.narmada@gmail.com
http://www.nitrkl.ac.in

Abstract. Automatic test case generation is a major problem in software testing. Evolutionary structural testing is an approach to automatically generate test cases that uses a Genetic Algorithm (GA) which is guided by the data flow dependencies in the program to search for test data to cover the def-use association. The Particle Swarm Optimization (PSO) approach is a swarm intelligence technique which can be used to generate test data automatically. We have proposed an algorithm to generate test cases using PSO for data flow testing. We have simulated both the evolutionary and swarm intelligence techniques. From the experiments it has been observed that PSO outperforms GA in 100% def-use coverage percentage.

Keywords: Software testing, Data flow testing, Genetic algorithm, Particle swarm optimization.

1 Introduction

Software testing has two main aspects: test data generation and application of a test data adequacy criterion. A test data generation technique is an algorithm that generates test cases. The test data adequacy criterion is a predicate that determines whether the testing process is finished. Several test data adequacy criteria have been proposed, such as control flow-based and data flow-based criteria.

An automated test data generator is a tool that assists the tester in creating test data. Test data generators can be categorized into three classes: random test data generators, structural-oriented test data generators and data specification generators. Random test data generators select random test data from the domain of input variables. Structural-oriented test data generators are based on path coverage, branch coverage, def-use coverage etc. Data specification generators select test data from program specification, in order to exercise features of the specification.

* Corresponding author.

S. Ranka et al. (Eds.): IC3 2010, Part II, CCIS 95, pp. 1–12, 2010.

Evolutionary structural testing is a search based technique that uses GA [1] which is guided by the data flow dependencies in the program, to search for test data to fulfill data flow path selection criteria namely the all-uses criterion.

However, GA [2,3,4,5,6,7] has started getting competition from other heuristic search techniques, such as the particle swarm optimization. Various works [8,9,10,11,12] show that particle swarm optimization is equally well suited or even better than genetic algorithms for solving a number of test problems. At the same time, a particle swarm algorithm is much simpler, easier to implement and has a fewer number of parameters that the user has to adjust than a genetic algorithm.

2 The Data Flow Analysis Technique

This section describes the all-uses criterion and the data flow analysis technique used to implement it. Firstly, some definitions used in describing this technique are presented.

The control flow of a program can be represented by a directed graph with a set of nodes and a set of edges. Each node represents a group of consecutive statements, which together constitute a basic block. The edges of the graph are then possible transfers of control flow between the nodes. A path is a finite sequence of nodes connected by edges. A complete path is a path whose first node is the start node and whose last node is an exit node. A path is def-clear with respect to a variable if it contains no new definition of that variable. Fig. 2 presents the flow graph of the example program, shown in Fig. 1, which determines the middle value of three given integers A, B and C.

Data flow analysis [13] focuses on the interactions between variable definitions (defs) and references (uses) in a program. Variable uses can be split into 'c-uses' and 'p-uses' according to whether the variable use occurs in a computation or a predicate. Defs and c-uses are associated with nodes, but p-uses are associated with edges. The purpose of the data flow analysis is to determine the defs of every variable in the program and the uses that might be affected by these defs, i.e. the def-use associations. Such data flow relationships can be represented by the following two sets: *dcu(i)*, the set of all variable defs for which there are def-clear paths to their cuses at node i; and *dpu(i,j)*, the set of all variable defs for which there are def-clear paths to their p-uses at edge (i,j).

Using information concerning the location of variable defs and uses, together with the basic static reach algorithm [14] the sets $dcu(i)$ and $dpu(i,j)$ can be determined . The basic static reach algorithm is used to determine two sets called $reach(i)$ and $avail(i)$. The set reach (i) is the set of all variable defs that "reach" node i. (A def of a variable x in node k is said to reach node i if there is a defclear path w.r.t. x from node k to node i). The set avail (i) is the set of all "available" variable defs at node i. It is the union of the set of global defs at node i together with the set of all defs that reach this node and are preserved through it. (Clearly any def of a variable in node i will not preserve any other def of the same variable). Using these two sets, the sets dcu(i) and dpu(i,j) are constructed from the formulae:

```
1    1   INTEGER A, B, C
2    1   READ (5,*) A, B, C
3    1   MID=C
4    1   IF (B.LT.C) THEN
5    2   IF (A.LT.B) THEN
6    3   MID=B
7    4   ELSE
8    4   IF (A.LT.C) THEN
9    5   MID=A
10   6   END IF
11   7   END IF
12   8   ELSE
13   8   IF (A.GE.B) THEN
14   9   MID=B
15  10   ELSE
16  10   IF (A.GT.C) THEN
17  11   MID=A
18  12   END IF
19  13   END IF
20  14   END IF
21  14   PRINT*,'MIDDLE VALUE= ', MID
22  14   END
```

Fig. 1. Example Program (The 1st column represents statement numbers, and the 2nd one represents block numbers)

$dcu(i) := reach(i) \cap c\_use(i)$, and
$dpu(i,j) := avail(i) \cap p\_use(i,j)$

where c-use(i) is the set of variables for which node i contains a global c-use, and p-use(i,j) is the set of variables for which edge (i,j) contains a p-use.

The all-uses criterion requires a def-clear path from each def of a variable to each use (c-use and p-use) of that variable to be traversed. It should be noted that, the all uses criterion includes all the members of the family of the data flow criteria, except the all-du-paths criterion. In other words, any complete path satisfying the all-uses criterion also satisfies the others. In order to determine the set of paths that satisfy the all-uses criterion, it is necessary to determine the def-use associations of program variables. As described above, such data flow relationships can be represented by the dcu and dpu sets.

The def-clear paths required to fulfil the *all-uses* criterion are constructed from the dcu and dpu sets. These paths are divided into two groups: dcu-paths and dpu-paths. In the dcu-paths list, each dcu-path is represented by: a def-node

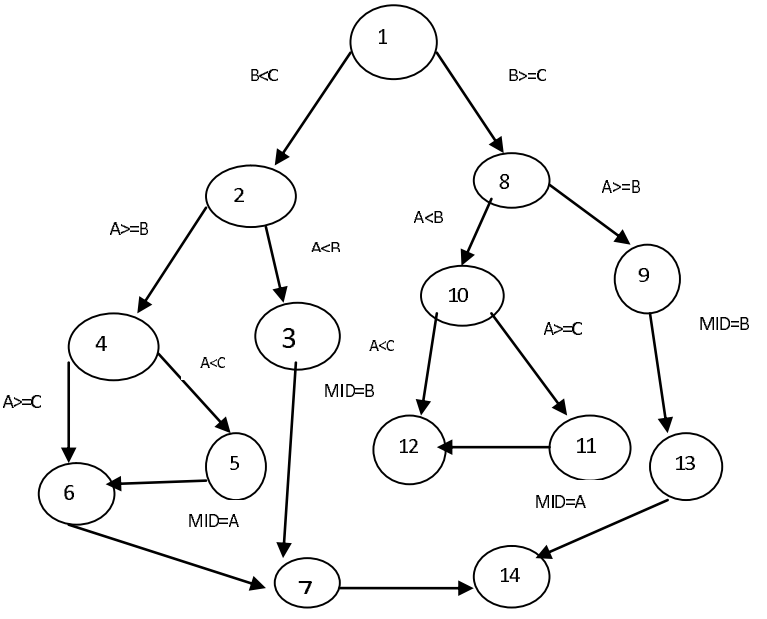

Fig. 2. Flow Graph for the Example Program given in Fig. 1

(a node containing a def of a variable), a c-use-node (a node containing a c-use of that variable), and the set of nodes that must not be included in that path (nodes containing other defs of that variable). These nodes are called killing nodes. In the dpu-paths list, each dpu-path is represented by: a def-node (node containing a def of a variable), p-use-edge (an edge having a p-use of that variable), and the set of killing nodes. Henceforth, the term 'def-use paths' will be used to mean the set of dcu-paths and dpu-paths together. Table 1 and 2 show the lists of the def-use paths of the example program.

Table 1. List of the DCU-paths of the Example Program given in Fig. 1

DCU Path No.	Variable	Def Node	C use Node	Killing Nodes
1	B	1	3	None
2	A	1	5	None
3	B	1	9	None
4	A	1	11	None
5	MID	1	14	3,5,9,11
6	MID	3	14	1,5,9,11
7	MID	5	14	1,3,9,11
8	MID	9	14	1,3,5,11
9	MID	11	14	1,3,5,9

Table 2. List of the DPU-paths of the Example Program given in Fig. 1

DPU Path No.	Variable	Def Node	P-Use Node	Killing Nodes
1	B	1	1-2	None
2	C	1	1-2	None
3	B	1	1-8	None
4	C	1	1-8	None
5	A	1	2-3	None
6	B	1	2-3	None
7	A	1	2-4	None
8	B	1	2-4	None
9	A	1	4-5	None
10	C	1	4-5	None
11	A	1	4-6	None
12	C	1	4-6	None
13	A	1	8-9	None
14	B	1	8-9	None
15	A	1	8-10	None
16	B	1	8-10	None
17	A	1	10-11	None
18	C	1	10-11	None
19	A	1	10-12	None
20	C	1	10-12	None

3 Proposed Work

The Genetic algorithm has been a significant milestone in automatic test data generation for dataflow testing, but the algorithm takes more generation to get the def-use coverage percentage. The Particle swarm optimization algorithm takes less generation to get the result.

This paper thus presents a comparative study and intends to find the efficient algorithm for automatic test data generation by performing simulations.

3.1 The Principles of Genetic Algorithm(GA)

Genetic algorithm is a meta-heuristic optimization technique that mimics the principles of the Darwinian theory of biological evolution. Its adequacy for solving non-linear, multi-modal, and discontinuous optimization problems has drawn the attention of many researchers and practitioners during the last decades. GAs generates a sequence of populations by using a selection mechanism and use crossover and mutation as search mechanisms. The principle behind GAs is that they create and maintain a population of individuals represented by chromosomes (essentially a character string analogous to the chromosomes appearing in DNA). These chromosomes are typically encoded solutions to a problem. The chromosomes then undergo a process of evolution according to rules of selection, mutation and reproduction.

The Genetic Algorithm to automatically generate test cases for the given program defines:

Step 1: generation = 0.
Step 2: Generate the initial population P_i of binary coded chromosomes C_i each representing a variables of the program.
Step 3: while ($coverage\_percent \neq 100$ *and* $No\_of\_Generation \leq Max\_Gen$)
Step 4: Generate Accumulated Def-Use coverage for each C_i using the evaluation Eq. (1)
Step 5: Get the mating pool ready by eliminating worst fit individuals and duplicating high fit individuals; using reproduction operator(cross over operator and mutation operator) reproduce offspring from the parent chromosomes and the new population is formed which will be current population P_i.
Step 6: generation = generation + 1.
Step 7: Extract a set of def-use paths covered by each test case.
Step 8: Calculate Accumulated Def-Use coverage for each C_i using the evaluation Eq. (1).
Step 9: If condition is not satisfied then go to step 3.

Evaluation Function

The algorithm evaluates each test case by executing the program with it as input, and recording the def-use paths in the program that are covered by this test case. (A test case is said to cover a def-use path, if it causes the program to traverse a path that has a sub path, which starts at the def-node and ends at the c-use node/p-use edge of the def-use path and does not pass through its killing nodes.) The fitness value evalC_i for each chromosome $C_i = (i = 1, \cdots, pop\_size)$ is calculated as follows:

$$eval(C_i) = \frac{Number\ of\ Def\_Use\ paths\ covered\ by\ C_i}{Total\ number\ of\ Def\_Use\ paths} \tag{1}$$

The fitness value is the only feedback from the problem for the GA. A test case which is represented by the chromosome C_i is considered effective if its fitness value eval $(C_i) > 0$.

Reproduction Operators

Crossover: The main purpose of crossover is to exchange information between two parent chromosomes to produce offspring for the next generation.
Mutation: The main aim of mutation is to introduce genetic diversity into the population

3.2 Particle Swarm Optimization(PSO)

Particle Swarm Optimization is an algorithm developed by Kennedy and Eberhart [8] that simulates the social behaviors of bird flocking or fish schooling and the methods by which they find roosting places, foods sources or other suitable habitat.

In the basic PSO technique, suppose that the search space is d-dimensional,

1. Each member is called particle, and each particle (i-th particle) is represented by d dimensional vector and described as $X_i = [X_{i1}, X_{i2},...,X_{id}]$.
2. The set of n particle in the swarm are called population and described as pop=$[X_1, X_2,...,X_d]$.
3. The best previous position for each particle (the position giving the best fitness value) is called particle best and described as $PB_i = [PB_{i1}, PB_{i2},...,PB_{id}]$.
4. The best position among all of the particle best position achieved so far is called global best and described as $GB_i = [GB_{i1}, GB_{i2},...,GB_{id}]$.
5. The rate of position change for each particle is called the particle velocity and described as $V_i = [V_{i1}, V_{i2},...,V_{id}]$.
   At iteration k the velocity for d-dimension of i-particle is updated by:

$$V_{id}^{k+1} = wV_{id}^{k} + c_1 r_1 (pb_{id}^{k} - x_{id}^{k}) + c_2 r_2 (gb_{id}^{k} - x_{id}^{k}) \tag{2}$$

   Where i= 1,2,..,n and n is the size of population, w is the inertia weight, c_1 and c_2 are the acceleration constants, and r_1 and r_2 are two random values in range [0,1].
6. The i-particle position is updated by:

$$x_{id}^{k+1} = x_{id}^{k} + V_{id}^{k+1} \tag{3}$$

PSO Algorithm to automatically generate test cases for the given program defines:

Step 1: (Initialization): Set the iteration number k=0. Generate randomly n particles, X_i , i = 1, 2,..., n, where $X_i = [X_{i1}, X_{i2},...,X_{id}]$. and their initial velocities $V_i = [V_{i1}, V_{i2},...,V_{id}]$. Evaluate the evaluation function for each particle eval (X_i) using Eq. (1). If the constraints are satisfied, then set the particle best $PB_i = X_i$, and set the particle best which give the best objective function among all the particle bests to global best GB. Else, repeat the initialization.
Step 2: Update iteration counter k=k+1.
Step 3: Update velocity using Eq. (2).
Step 4: Update position using Eq. (3).
Step 5: Update particle best:
If $eval_i(X_i^k) > eval_i(PB_i^{k-1})$ then
$PB_i^k = X_i^k$
Else $PB_i^k = PB_i^{k-1}$

Step 6: Update global best:
$eval(GB^k) = max(eval_i(PB_i^{k-1}))$
If $eval(GB^k) > eval(GB^{k-1})$ then
$GB^k = GB^k$
Else $GB^k = GB^{k-1}$

Step 7: (Stopping criterion): If the number of iteration exceeds the maximum number iteration or accumulated coverage is 100% then stop, otherwise go to step 2.

4 Comparison between GA and PSO

The Genetic Algorithm (GA) is an evolutionary optimizer that takes a sample of possible solutions (individuals) and employs mutation, crossover, and selection as the primary operators for optimization.
Most of evolutionary techniques have the following procedure:

1. Random generation of an initial population.
2. Reckoning of a fitness value for each subject.
3. Reproduction of the population based on fitness values.
4. If requirements are met, then stop. Otherwise go back to 2.

From this procedure, we can learn that PSO shares many common points with GA. Both algorithms start with a group of randomly generated population and both algorithms have fitness values to evaluate the population. Both algorithms update the population and search for the optimum with random techniques. Compared with genetic algorithms (GAs), PSO does not have genetic operators like crossover and mutation. Particles update themselves with the internal velocity. They also have memory, which is important to the algorithm. Also, the information sharing mechanism in PSO is significantly different: In GAs, chromosomes share information with each other. So the whole population moves like one group towards an optimal area even if this move is slow. In PSO, only GB gives out the information to others. It is one-way information sharing mechanism. The evolution only looks for the best solution. Compared with GA, all the particles tend to converge to the best solution quickly in most cases.

When comparing the run-time complexity of the two algorithms, we should exclude the similar operations (initialization, fitness evaluation, and termination) from our comparison because similar operations give same run time complexity. We exclude also the number of generations, as it depends on the problem complexity and termination criteria (experimentally, PSO has lower number of generations). Therefore, we will make our calculations for the main loop of the two algorithms. We consider the most time-consuming processes (recombination in GA as well as velocity and position update in PSO).

For GA, if the new generation replaces the older one, the recombination complexity is O(q), where q is group size for tournament selection. In our case, q equals the Selection rate*n, where n is the size of population. But if the replacement strategy depends on to the fitness of the individual, a sorting process is needed to determine which individuals to be replaced by which new individuals. This sorting is important to guarantee the solution quality. Another sorting process is needed any way to update the rank of the individual at the end of each generation. Therefore, as the quick sort complexity ranges from $O(n^2)$ to $O(n \log_2 n)$ the recombination complexity is $O(n^2)$ to $O(n \log_2 n)$.

For PSO, the velocity and position update processes complexity is O (n) as there is no need for sorting. The algorithm operates according to equations (2) and (3) on each individual (particle).

From the above discussion, GA's complexity is larger than that of PSO. Therefore, PSO is simpler and faster than GA.

5 Simulation Results

MATLAB is used as simulation tool.Input to the PSO algorithm is instrumented version of the program, list of def-use path to be covered, number of program input variables, population size, and maximum number of iteration.Output is the set of test cases for program and the set of def-use path covered by each test case. The effectiveness of the proposed PSO is compared with GA. A set of 14 small FORTRAN programs is used in the experiments.

Table 3. Comparison between GA technique and PSO Technique

			No.of Generation		DEf-Use Coverage %	
Program No.	No.of Variable	Population size	GA	PSO	GA	PSO
1	3	6	17	2	100	100
2	3	8	10	3	100	100
3	4	8	19	6	100	100
4	3	6	21	17	100	100
5	5	10	9	3	100	100
6	4	6	17	9	100	100
7	4	8	13	7	100	100
8	3	8	12	5	85	90
9	4	8	14	10	90	96
10	4	6	19	12	100	100
11	3	8	21	15	82	95
12	4	6	18	10	100	100
13	3	8	24	18	81	92
14	3	6	19	10	100	100

Table 3 shows the result of applying the GA technique and PSO technique to 10 programs. As can be seen PSO technique outperformed the GA technique in all programs. In these programs, the PSO technique required less number of generations than GA technique to achieve same def-use coverage percentage.

In each iteration of PSO, we find the GB (gbest), which gives the single optimal solution. The PSO moves towards the near global optimal solution in each iteration. Thereby, PSO takes less iterations whereas in GA, in each generation the three basic opertions namely selection, crossover and mutation are performed. So GA takes more generations to reach towards the solution in comparison to PSO.

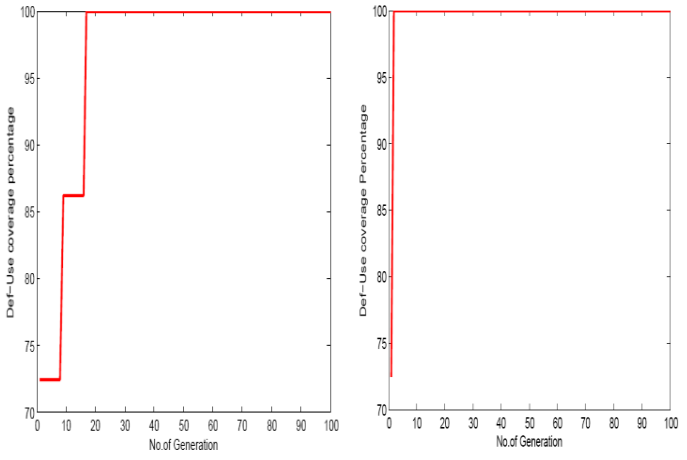

Fig. 3. (a)The graphs showing plot between Def-Use Coverage Percentage vs. No. of Generations of GA technique of 1st Program (b) The graphs showing plot between Def-Use Coverage Percentage vs. No. of Generations of PSO technique of 1st Program

It should be noted that, in the cases where less than 100% coverage is achieved, the program included some def-use paths that cannot be covered by any test case due to the existence of infeasible paths.

In Fig. 3(a) shows the graph between the Def-Use Coverage percentage vs. No.of Generations of GA technique of 1st program. Fig. 3(b) shows the graph between the Def-Use Coverage percentage vs. No.of Generations of PSO technique of 1st program. In the graph it is viewed that GA technique has taken 17 generations to get 100% def-use coverage but PSO technique has taken 2 iterations to get the same result. This improvement in PSO occurs due to the reasons mentioned above.

6 Comparison with Related Work

Li et al. [11] applied particle swarm optimization to all path test data generation. They constructed a new fitness function for all path test data generation. In the process of all path test data generation, the execution frequencies of all paths are registered to apply to further tests. If the frequency is equal to zero, this indicates that the path is not traveled in this generation. In this situation they used the single path test data automatic generation to generate test data for the path. If it is not found then the path is probably unreachable. If frequency data is relatively small, the situation shows that this path is infrequent one. If the frequency data is relatively big, then the situation is considered as normal.

Khushboo et al. [10] described the application of the discrete quantum particle swarm optimization (QPSO) to the problem of automated test data generation. A discrete quantum particle swarm optimization algorithm based on the concept of quantum computing. They had studied the role of the critical QPSO

parameters on test data generation performance and based on the observation an adaptive version (AQPSO) had been designed. Its performance compared with QPSO. They used the branch coverage as their test adequacy criteria.

Andreas et al. [9] described the application of Comprehensive Learning Particle Swarm Optimizer (CL-PSO). They applied a new learning strategy, where each particle learns from different neighbors for each dimension separately which dependent on its assigned learning rate. This happenes until the particle does not achieve any further improvements for a specific number of iterations called the refreshing gap. Finally it yields a reassignment of particles. They described the design of the test system that was used to carry out experiments with 25 artificial and 13 industrial test objects that exhibit different search space properties. Both particle swarm optimization and genetic algorithms were used to automatically generate test cases for the same set of test objects. They used the branch coverage as their test adequacy criteria. The results of the experiments showed that particle swarm optimization is competitive with genetic algorithms and even outperforms them for complex cases.

Andreas et al. [9] and Khushboo et al. [10] used the branch coverage as their test adequacy criteria. Once such a criterion has been chosen, test data must be selected to fulfill the criterion. One way to accomplish this is to select paths through the program whose elements fulfill the chosen criterion, and then to find the input data which would cause each of the chosen paths to be selected. Li et al. [11] used the *all-paths* selection criterion, which require all program paths to be selected. Using such path selection criteria as the basis for test data selection criteria presents two distinct problems. The first problem is that traversing all paths does not guarantee that all errors will be detected. The second problem is that programs with loops may have an infinite number of paths, and thus, the all-paths criterion must be replaced by a weaker one which selects only a subset of the paths.

But in our approach, we are using *all-uses* as test adequacy criterion which focuses on how the variables are bound to values, and how these variables are used. Rather than selecting program paths based solely on the control structure of a program, the data flow *all-uses* criteria keep track of input variables through a program, following them as they are modified, until they are ultimately used to produce output values.

7 Conclusion and Future Work

We have developed an algorithm for generating test cases using PSO. The PSO technique accepts an instrumented version of the program to be tested,the list of def-use paths to be covered, the number of input variables. Also, it accepts the the population size, maximum number of iterations, values of inertia factor, self confidence and swarm confidence. The algorithm produces a set of test cases and the set of def-use paths covered by each test case.

Experiments have been carried out to evaluate the effectiveness of the proposed PSO compared to the GA technique. The results of these experiments

show that the PSO technique outperforms the GA technique in all 14 programs used in the experiment. In four programs, the PSO reached higher coverage percentage in fewer generation than the GA technique.

Our future work will be to study the test case generation using ant colony optimization technique for data flow testing.

References

1. Girgis, M.R.: Automatic test data generation for data flow testing using genetic algorithm. Journal of Universal Computer Science 11(6), 898–915 (2005)
2. Pargas, R.P., Horrold, M.J., Peck, R.R.: Test data generation using genetic algorithm. The Journal of Software Testing,Verification and Reliability (1999)
3. Michael, C.C., McGraw, G., Schatz, M.A.: Generating software test data by evolution. IEEE Transactions on Software Engineering 27(12), 1085–1110 (2001)
4. Pei, M., Goodman, E.D., Gao, Z., Zhong, K.: Automated software test data generation using genetic algorithm. Technical report, GARGE of Michigan State University (1994)
5. Jones, B.F., Sthamer, H.H., Eyres, D.E.: Automatic structural testing using genetic algorithms. Software Engineering Journal 8(9), 299–306 (1996)
6. Roper, M., Maclean, I., Brooks, A., Miller, J., Wood, M.: Genetic algorithm and the automatic generation of test data. Technical report, University of Strathelyde (1995)
7. Watkins, A.E.L.: A tool for automatic generation of test data using genetic algorithm. In: Software Quality Conference, Dundee, Scotland (1995)
8. Kennedy, J., Eberhart, R.: Particle swarm optimization. In: IEEE International Conference on Neural Networks, pp. 1942–1948. IEEE Press, Los Alamitos (1995)
9. Windisch, A., Wappler, S., Wegener, J.: Applying paricle swarm optimization to software testing. In: GECCO, London, England, United Kingdom. ACM, New York (2007)
10. Agrawal, K., Srivastava, G.: Towards software test data generation using discrete quantum particle swarm optimization. In: ISEC, Mysore, India (February 2010)
11. Li, A., Zhang, Y.: Automatic generating all-path test data of a program based on pso. In: World Congress on Software Engineering. IEEE, Los Alamitos (2009)
12. Eberhart, R.C., Kennedy, J.: A new optimizer using particle swarm theory. In: 6th International Symposium on Micromachine Human Science, pp. 39–43 (1995)
13. Rapps, S., Weyuker, E.J.: Selecting software test data using data flow information. IEEE Transactions on Software Enggineering 11(4), 367–375 (1985)
14. Allen, F.E., Cocke, J.: A program data flow analysis procedure. Communication of the ACM 19(3), 137–147 (1976)