Automatic Data Flow Test Paths Generation using the Genetical Swarm Optimization Technique

Moheb R. Girgis
Department of Computer Science,
Faculty of Science, Minia University,
El-Minia, Egypt

Ahmed S. Ghiduk
Dept. of Mathematics and CS,
Faculty of Science,
Beni-Suef University, Egypt
College of Computers and IT,
Taif University, Saudi Arabia

Eman H. Abd-Elkawy
Dept. of Mathematics and CS,
Faculty of Science,
Beni-Suef University, Egypt

ABSTRACT
Path testing requires generating all paths through the program to be tested, and finding a set of program inputs that will execute every path. The number of possible paths in programs containing loops is infinite, and so it is very difficult, if not impossible, to test all of them. Path testing can be relaxed by selecting a subset of all executable paths that fulfill a certain path selection criterion and finding test data to cover it. The automatic generation of such test paths leads to more test coverage paths thus resulting in efficient and effective testing strategy. This paper presents a genetical swarm optimization (GSO) based technique, which effectively combines a genetic algorithm (GA) based technique and a particle swarm optimization (PSO) based technique, for automatic generation of a set of test paths that cover the all-uses criterion. Experiments have been carried out to evaluate the effectiveness of the proposed GSO approach in test paths generation compared to the GA and PSO approaches.

Keywords
Software testing, Automatic test path generation, Data flow testing, Genetic Algorithms, Particle Swarm Optimization, Genetical Swarm Optimization

1. INTRODUCTION
Path testing requires that every path through a program to be executed at least once. However, the number of possible paths in programs containing loops is infinite, and so it is very difficult, if not impossible, to test all of them. This problem can be solved by selecting a subset of all executable paths that fulfill certain path selection criterion. The automatic generation of such test paths leads to more test coverage paths thus resulting in efficient and effective testing strategy.

Several path generation methods have been proposed. For example, Bertolino and Marre [1] provided an algorithm that finds a path that covers every arc in a given program control flow graph (CFG). Most other research studies have focused on the automatic generation of a basis set of paths, which is a set of independent test paths, where the number of test paths in this set equals to the cyclomatic complexity of the program defined by McCabe [2], (see e.g., [3], [4], [5], [6], [7]).

Genetic algorithms (GAs) have been successfully used in software testing activities such as test data generation, (see e.g., [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19]). But very little attention has been paid to use GAs in path testing [20]. For example, Bint and Site [21] developed a variable length GA for identifying the most error prone path clusters in a program; and Ghiduk et al. [22] introduced a strategy for automatically generating a set of basis test paths using a variable length GA.

Particle Swarm Optimization (PSO) is an intelligent technology first presented in 1995 by Kennedy and Eberhart [23], and it was developed under the inspiration of behavior laws of bird flocks, fish schools and human communities. PSO has been successfully used in software testing activities such as test data generation, (see e.g., [24], [25], [26], [27], [28], [29]).

The Genetical Swarm Optimization (GSO) [30] is a hybrid evolutionary technique that exploit in the most effective way the uniqueness and peculiarities of the PSO and GAs. This algorithm is essentially, as PSO and GA, a population-based heuristic search technique, which can be used to solve combinatorial optimization problems, modeled on the concepts of natural selection and evolution (GA), but also based on cultural and social rules derived from the analysis of the swarm intelligence and from the interaction among particles (PSO). So far as the authors are aware, no research work in the area of using PSO or GSO in automatic generation of test paths has been reported.

This paper presents a GSO-based technique for automatic generation of a set of test paths that cover the all-uses criterion [31]. Since the GSO combines the GA and PSO techniques, the paper, firstly, presents two proposed GA-based and PSO-based techniques for test paths generation. These two techniques conduct their search by constructing new paths from previously generated paths that are evaluated as effective test paths. In each iteration of the proposed GSO algorithm, the population is divided into two parts and they are evolved with the PSO and GA techniques respectively. They are then recombined in the updated population, that is again divided randomly into two parts in the next iteration for another run of genetic or particle swarm operators. This evolutionary process not only improves the individuals score for natural selection of the fitness or for good-knowledge sharing, but for both of them at the same time.

The paper is organized as follows: Section 2 describes the data flow analysis technique used to implement the all-uses criterion. Sections 3 and 4 describe the proposed GA-based and PSO-based techniques for test paths generation, respectively. Section 5 describes the proposed GSO-based technique for test paths generation, which combines these two techniques. Section 6 presents the results of the experiments that are conducted to evaluate the effectiveness of the proposed GSO-based technique compared to the GA-based and PSO-based test paths generation techniques.

2. DATA FLOW ANALYSIS

Data flow analysis focuses on the interactions between variable definitions (defs) and references (uses) in a program, i.e. the def-use associations. Data flow analysis techniques use a CFG representation of the program under test to compute def-use associations. In a CFG, each node represents a statement, and the edges are 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. In this work, a reduced form of the CFG, called the DD-graph, is used, in which each edge represents a DD (decision to decision) path. Figure 2 shows the DD-graph that corresponds to the CFG of the example program shown in Figure 1. Table 1 shows the DD-paths that correspond to the edges of the DD-graph shown in Figure 2.

The all-uses criterion is one of the data flow testing criteria proposed by Rapps and Weyuker [32]. It requires a def-clear path from each def of a variable to each use of that variable to be traversed. The def-clear paths required to satisfy the all-uses criterion, called def-use paths, are constructed from the def-use associations of program variables by using the technique described in [32].

```
1.  using System;
2.  using System.IO;
3.  public class prog5
4.  {
5.     static void Main()
6.     {
7.        int a, b, c, n;
8.        a = Int32.Parse(Console.ReadLine());
9.        b = Int32.Parse(Console.ReadLine());
10.       if (a < 5)
11.       {
12.          c = a;
13.       }
14.       else
15.       {
16.          c = b;
17.       }
18.       n = c;
19.       while (n <= 8)
20.       {
21.          if (b > c)
22.          {
23.             c = 3;
24.          }
25.          else
26.          {
27.             n = n + c;
28.          }
29.          n = n + 1;
30.       }
31.    Console.WriteLine(" {0} {1} {2}", a, b, n);
32.    }//end main
33. }//end class
```

Figure 1: Example program

3. GA-BASED TECHNIQUE FOR TEST PATHS GENERATION

This section describes a proposed GA for automatic test path generation [33]. The algorithm searches for test paths that satisfy the all-uses criterion. In the case of programs containing loops, the proposed technique generates paths according to the ZOT-subset criterion: "Each loop in a program is iterated zero, one, and two times in execution" [34].

3.1 Representation

The algorithm uses a binary vector as a chromosome to represent the edges in the DD-graph of the program under test. The length, L, of the vector equals to the number of the edges of the DD-graph of the program under test, including two extra edges representing the entry and exit edges, plus the number of edges contained in loops, as those edges are represented twice. This representation guarantees that the paths generated for programs containing loops satisfy the ZOT-subset criterion. For example, the main edges of the DD-graph of the example program, shown in Figure 2, are: e1, e2 … e9, in addition to the entry and exit edges, and the edges contained in the While loop are e6, e7, e8, and e9. A copy of the loop edges are added after the last edge, e9, with numbers starting from 10, i.e., they will be e10, e11, e12, and e13. So, the chromosome length becomes 15, and it takes the following form:

e0	e1	e2	e3	e4	e5	e6	e7	e8	e9	e10	e11	e12	e13	e14

where e0 and e14 are the entry and exit edges, respectively, and the shaded genes, e10 to e13, represent a copy of the loop edges, e6 to e9.

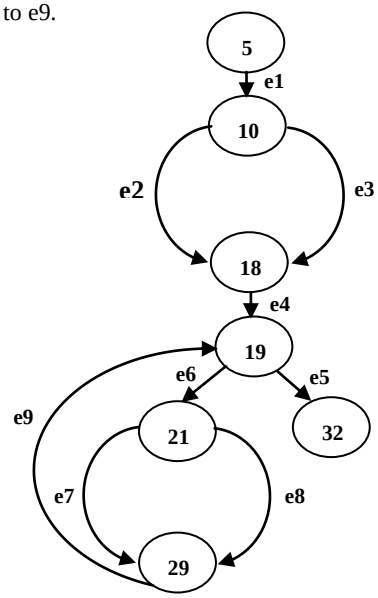

Figure 2: The DD-graph of the example program

Table 1: The edges of the DD-graph shown in Figure 2 and the corresponding DD-paths

Edge	DD-Path
e1	5 6 7 8 9 10
e2	10 14 15 16 17 18
e3	10 11 12 13 18
e4	18 19
e5	19 31 32
e6	19 20 21
e7	21 25 26 27 28 29
e8	21 22 23 24 29
e9	29 30 19

Consider an example chromosome: 110111101110111. Using the above representation, this chromosome represents the following edges:

e0, e1, e3, e4, e5, e6, e8, e9, e10, e12, e13, e14

These edges form the following connected path:

e0, e1, e3, e4, e6, e8, e9, e10, e12, e13, e5, e14

By replacing each edge with its corresponding DD-path, the path in terms of the program statements becomes as follows:

5,6,7,8,9,10,11,12,13,18,19,20,21,22,23,24,29,30,19,20,21,22,23,24,29,30,19,31,32

3.2 Initial Population

As mentioned above, each chromosome (as a test path) is represented by a binary string of length L. The algorithm randomly generates POPSIZE L-bit strings to represent the initial population, where POPSIZE is the population size. The appropriate value of POPSIZE is experimentally determined. Each test path in the generated population must satisfy the connectivity condition, i.e., it consists of a sequence of connected edges. If the generated chromosome does not represent a connected path, the algorithm discards it.

3.3 Evaluation Function

The algorithm evaluates each test path by determining the def-use paths in the program that are covered by this test path. (A test path is said to cover a def-use path, if it includes a subpath, which starts at the def-node and ends at the use node of the def-use path and does not pass through its killing nodes.) The fitness value fitness_value(vi) for each chromosome vi (i = 1, …, POPSIZE) is calculated as follows [16]:

$$\text{fitness\_value}(v_i) = \frac{\text{no. of def–use paths covered by } v_i}{\text{total no. of def–use paths}} \quad (1)$$

3.4 Selection

After computing the fitness of each test path in the current population, the algorithm uses the roulette wheel method [35] to select test paths from the effective members of the current population that will be parents of the new population. If none of the members of the current population was effective, all the members of current population are considered the parents of the new population.

3.5 Recombination

Crossover and mutation operators create new individuals from the selected parents to form a new population.

Crossover: It operates at the individual level with a pre-determined probability pc. During crossover, two parents (chromosomes) exchange substring information (genetic material) at a random position in the chromosome to produce two new strings (offspring).

Mutation: It is performed on a gene-by-gene basis. Mutation always operates after the crossover operator, and changes each gene with the pre-determined probability pm. Every gene (in all chromosomes in the whole population) has an equal chance to undergo mutation. A gene is mutated by replacing its corresponding edge with another edge from its siblings (edges with the same parent are called siblings).

Figure 3 shows the overall GA-based test paths generation algorithm.

4. PSO-BASED TECHNIQUE FOR TEST PATHS GENERATION

This section describes the proposed PSO-based algorithm for automatic test path generation. The algorithm searches for test paths that satisfy the all-uses criterion. In the case of programs containing loops, the proposed technique generates paths according to the ZOT-subset criterion.

```
/* A GA-based algorithm to automatically generate test paths
that cover the all-uses criterion for a given program */
Input:
    The program to be tested P;
    List of program def-use paths to be covered;
    List of program edges;
    DD-graph of P
    Population size;
    Maximum no. of generations (MAXGENS);
    Probability of crossover pc;
    Probability of mutation  pm;
Output:
    Set of test paths, and the set of def-use paths covered by
    each test path;
Begin
    Step 1: Initialization
        Initialize the def-use coverage vector to zeros;
        Randomly create Initial_Population of chromosomes
        (test paths) such that each generated test path must
        satisfy the connectivity condition;
        Current_population ← Initial_Population;
        def-use coverage percent ← 0
        accumulated def-use-coverage percent ← 0
        No_Of_Generations ← 0;
        nPaths← 0;
    Step 2: Generate test paths
        While (Coverage_Percent ≠100 and
                    No_Of_Generations ≤ MAXGENS) do
        Begin
          nEffective ← 0;
          For each member of current population do
            Convert the current chromosome to the
            corresponding path;
            Evaluate the current test path;
            If (some def-use paths are covered) then
              nPaths ← nPaths + 1;
              Add effective test path to set of test paths for P;
              Update the def-use coverage vector;
              Update accumulated def-use- coverage;
              nEffective ← nEffective + 1;
            End If
          End For;
          If (nEffective > 0) then
              Select set of parents of new population from
              effective members of current population
              using roulette wheel method
          Else
              Set of parents of new population ←
                                  Current_Population;
          End If;
          Apply crossover, mutation operators to create
          New_Population such that each new offspring must
          satisfy the connectivity condition;
          Current_Population ← New_Population;
          Increment No_Of_Generations;
        End While
    Step 3: Produce output
        Return set of test paths for P and set of def-use paths
        covered by each test path;
End.
```

Figure 3: The proposed GA-based test paths generation algorithm

4.1 Principles of PSO

PSO is a robust stochastic optimization technique, which is inspired from the movement and intelligence of swarms. PSO applies the concept of social interaction to problem solving. It was developed in 1995 by James Kennedy and Russell

Eberhart [23]. It uses a number of particles that constitute a group moving around in the search space looking for the best solution. It imitates the bird from a flock which is nearest to the food. Each particle is treated as a point in an N-dimensional space which adjusts its "flying" according to its own flying experience as well as the flying experience of other particles. All particles have fitness values, evaluated through the fitness function and velocities. The two variables which are iteratively changed in PSO algorithm are the following ones:

- *pbest* (personal best): each particle keeps track of its coordinates in the solution space which are associated with the best solution (fitness) that has achieved so far by that particle;
- *gbest* (global best): another best value that is tracked by the PSO is the best value obtained so far by any particle in the neighborhood of that particle;

Each particle tries to modify its position using the following information: the current positions, the current velocities, the distance between the current position and pbest, the distance between the current position and gbest.

The fitness function has to be customized for each application of the PSO. The updates of the particles' position and velocity are made using Eqs. (2) and (3), respectively, [36]:

$$v_i^{k+1} = K * [v_i^k + c_1 * r_1 * (pbest_i - p_i^k) + c_2 * r_2 * (gbest - p_i^k)] \quad (2)$$

$$K = \frac{2}{\left|2-\varphi-\sqrt{\varphi^2-4\varphi}\right|}, \text{ where } \varphi = c_1 + c_2, \varphi > 4,$$

$$p_i^{k+1} = p_i^k + v_i^{k+1} \quad (3)$$

where:

- v_i^k: velocity of particle *i* at iteration *k*
- K: a Constriction factor, which is necessary to insure the convergence of the PSO algorithm [37]
- c1, c2: learning factors, which are constants with values between 0 and 4: their best values are established, usually, experimentally
- r_1, r_2: uniformly distributed random number between 0 and 1
- p_i^k: current position of particle *i* at iteration *k*
- $pbest_i$: pbest of particle *i*
- gbest: gbest of the group

4.2 Representation

In the proposed PSO algorithm, the position vector of a particle represents the edges of a path in the DD-graph of the program under test, as the chromosome in the proposed GA algorithm.

The proposed PSO algorithm uses two forms for the position vector, a binary form, called binary position vector, and a corresponding integer form, called actual position vector. In the binary position vector, the value 1 represents an existing edge, and the value 0 represents a missing edge. The actual position vector is constructed as follows: All 1's in the binary position vector are represented in the actual position vector by the indices of the corresponding edges, then the vector is padded with a number of -1's equals to the number of 0's in the binary position vector. The value -1 is used to represent missing edges because edges' indices are positive integers. The velocity vector of a particle consists of L integers from the range [0, L].

Consider an example particle whose binary position vector is: 110111101110111. Using the above representation, this position vector represents the following edges: e0, e1, e3, e4, e5, e6, e8, e9, e10, e12, e13, e14. Thus, the corresponding actual position vector will be:

0, 1, 3, 4, 5, 6, 8, 9, 10, 12, 13, 14, -1, -1, -1.

These edges form the following connected path:

e0, e1, e3, e4, e6, e8, e9, e10, e12, e13, e5, e14

By replacing each edge with its corresponding DD-path, the path in terms of the program statements becomes as follows:

5,6,7,8,9,10,11,12,13,18,19,20,21,22,23,24,29,30,19,20,21,22,23,24,29,30,19,31,32

4.3 Initial Population

As mentioned above, the position vector of each particle (as a test path) is represented by two vectors, binary position vector and actual position vector, both of length L. The algorithm randomly generates POPSIZE L-bit strings to represent the binary position vectors of the initial population, where POPSIZE is the population size. The elements 0,1and L-1 of the binary position vector of a particle are set to 1's, because they represent the entry, first and exit edges, and must be included in any path. The appropriate value of POPSIZE is experimentally determined. Then, the algorithm constructs, for each generated binary position vector, the corresponding actual position vector, as described above. Each test path in the generated population must satisfy the connectivity condition. If the generated position vector does not represent a connected path, the algorithm discards it and generates another one.

Also, the algorithm randomly generates POPSIZE velocity vectors for the initial population. The elements of each velocity vector are random integers between 0 and L-1, except the elements 0, 1 and L-1, which are set to the values: 0, 0 and L, respectively. This setting ensures that the actual position vector update operation results in a path that includes the entry, first and exit edges of any complete path.

4.4 Updating Position and Velocity Vectors

In the test path generation problem, the update operations of the velocity and position of a particle are performed on its actual position vector and velocity vector, which are coded as integer sequences, as described in Section 4.2, rather than real number vectors. In the proposed algorithm, the addition and subtraction operators in the original velocity and position update equations are modified to adapt to the test path generation problem domain.

The proposed algorithm uses two operators, ***add-position-and-velocity operator***, and ***subtract-two-positions operator***. These operators are used in creating new individuals from the current population to form a new population.

Subtract-two-positions operator: The velocity vector update equation, Eq. 2, includes two subtraction operations between positions. The subtract-two-positions operator is defined as follows:

$$p_i' - p_i = \begin{cases} p_i' - p_i & if\ p_i' - p_i > 0 \\ (p_i' - p_i) + (L-3) & if\ p_i' - p_i \le 0 \end{cases} \quad (4)$$

where $\{p_i\}_{i=0}^{L-1}$ and $\{p_i'\}_{i=0}^{L-1}$ are two position vectors.

Add-position-and-velocity operator: The position vector update equation, Eq. 3, adds the new velocity to the old position. The add-position-and-velocity operator is defined as follows:

$$p_i + v_i = \begin{cases} p_i + v_i & if\ p_i + v_i < L \\ ((p_i + v_i)\%(L-1)) + 2 & if\ p_i + v_i \geq L \end{cases} \quad (5)$$

where $\{p_i\}_{i=0}^{L-1}$ and $\{v_i\}_{i=0}^{L-1}$ are position and velocity vectors, respectively.

4.5 Evaluation Function

The proposed PSO algorithm uses the same fitness function, Eq. 1, of the proposed GA algorithm to evaluate the fitness value of each particle (test path).

Figure 4 shows the overall PSO-based test paths generation algorithm.

5. GSO-BASED TECHNIQUE FOR TEST PATHS GENERATION

This section presents the proposed GSO-based technique for automatic generation of a set of test paths that cover the all-uses criterion. The proposed GSO is characterized by a strong co-operation of the GA-based and PSO-based test paths generation techniques, described in the previous two sections, since it maintains the integration of the two techniques for the entire run. In each iteration of the proposed GSO, the population is divided into two parts and they are evolved with the two techniques respectively. They are then recombined in the updated population, that is again divided randomly into two parts in the next iteration for another run of genetic or particle swarm operators.

The splitting of the population is done according to a hybridization coefficient (hc) [30], which expresses the part of the population that is evolved in each iteration with GA. In the proposed GSO algorithm, hc = 0 means the procedure is a pure PSO (the whole population is evolved according to PSO operators), hc = POPSIZE means pure GA (the whole population is evolved according to GA operators), while 0 < hc < POPSIZE means that hc individuals from the population is evolved by GA, while the rest with PSO technique. It should be noted that, at the beginning of each iteration a new value of hc is randomly generated. Figure 5 shows the way of splitting the population into PSO and GA populations during the iterations.

In the proposed GSO approach, if the updated position vector of a particle in the PSO population, or the offspring generated by applying the crossover and mutation operators in the GA population, does not satisfy the connectivity condition, the algorithm tries to repair it, i.e. makes it connected, by adding and/or removing one or more edges to/from it. If the repair operation failed, the updated position vector or the generated offspring will be discarded, and replaced by the original position vector or chromosome in the new population. Figure 6 shows the proposed GSO-based test paths generation algorithm.

Figure 7 shows part of the report produced by the proposed GSO when it is applied to the example program. This report shows that, in 3 generations, the GSO has generated 7 test paths that covered 100% of the program def-use pairs.

```
/* A PSO algorithm to automatically generate test paths that cover the all-uses criterion for a given program */
Input:
    The program to be tested P;
    List of program def-use paths to be covered;
    List of edges and DD-graph of P;
    Population size;
    Maximum no. of generations (MAXGENS);
    c1, c2: learning factors
Output:
    Set of test paths and set of def-use paths covered by each test path;
Begin
    Step 1: Initialization
        Initialize the def-use coverage vector to zeros;
        Randomly create Initial_Population of particles with random positions (test paths) and velocities for each particle such that each generated test path must be connected;
        Current_population ← Initial_Population;
        def-use coverage percent ← 0
        accumulated def-use-coverage percent ← 0
        No_Of_Generations ← 0;
        nPaths←0;
    Step 2: Generate test paths
        For each particle p with position x_p in
                                        Current_population do
          Convert x_p to the corresponding path;
          Evaluate the current test path (calculate its fitness value);
          If (fitness(x_p) is better than fitness(pbest_p)) then
            pbest_p ← x_p.
          If (some def-use paths are covered) then
            nPaths ← nPaths + 1;
            Add effective test path to set of test paths for P;
            Update the def-use coverage vector;
            Update accumulated def-use-coverage percent;
          End If
        End For;
        Identify the particle in the swarm with the best pbest, gbest ← best pbest.
        While (Coverage_Percent ≠100 and
                        No_Of_Generations ≤ MAXGENS) do
          For each particle p in Current_population do
            Update velocity v_p and position x_p;
            // Check connectivity
            Convert updated x_p to the corresponding path pth;
            If (pth is connected) then
              Add new particle to new population
            Else
              Add the original particle to the new population
            End If;
          End for
          For each particle p with position x_p in new population do
            Convert x_p to the corresponding path;
            Evaluate the current test path (calculate fitness value);
            If (fitness(x_p) is better than fitness(pbest_p)) then
              pbest_p ← x_p.
            If (some def-use paths are covered) then
              nPaths ← nPaths + 1;
              Add effective test path to set of test paths for P;
              Update the def-use coverage vector;
              Update accumulated def-use-coverage percent;
            End If
          End For;
          Identify the particle in the swarm with the best pbest,
          If (best pbest is better than gbest) then
            gbest ← best pbest;
          Current_population ← new population;
          Increment No_Of_Generations;
        End While
    Step 3: Produce output
        Return set of test paths for P and set of def-use paths covered by each test path;
End.
```

Figure 4: The proposed PSO-based test paths generation algorithm

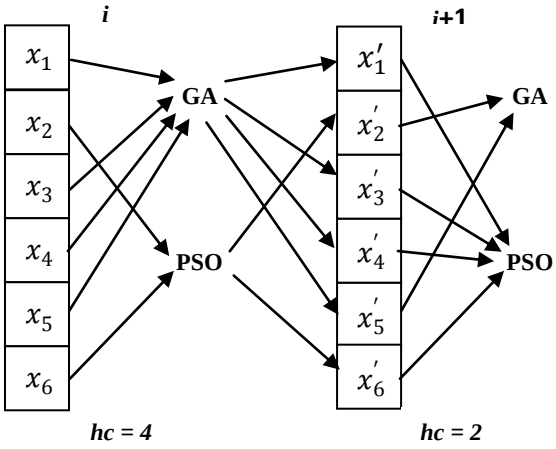

Figure 5: Splitting the population, according to hc, into PSO and GA populations during the iterations.

The set of paths generated by the proposed GSO can be passed to a test data generation tool to find program inputs that will execute them to complete the data flow paths testing of the program under test.

6. EXPERIMENTS

This section presents the results of the experiments that have been conducted to evaluate the effectiveness of the proposed metaheuristic paths generation techniques, GA, PSO, and GSO. The materials of the experiments were 15 small C# programs. The used GA and PSO parameters were as follows: MAXGENS=100, pc=0.8, pm=0.15, c1=3, c2=3, and POPSIZE=4.

Table 2 shows the results of applying the GSO, PSO and GA techniques to the 15 programs. As can be seen from the table, the GSO technique required less number of generations to cover all def-use paths than the GA technique in 14 out of the 15 programs, and the PSO technique in 12 out of the 15 programs. For example, for program P#12, the GA technique required 28 generations to cover 100% of the def-use paths, and the PSO technique required 8 generations, while the GSO technique required only 6 generations. In program P#4, all the three techniques completed the MAXGENS generations and covered only 88.89% of all def-use paths. It should be noted that, in the cases where less than 100% coverage is achieved, the programs included some def-use paths that cannot be covered by any test paths due the existence of infeasible paths. In program P#13 the GSO technique covered all def-use paths in 11 generations, the PSO technique covered all def-use paths in 12 generations, while the GA technique completed the MAXGENS generations and covered only 91.11% of all def-use paths.

Figure 8 shows a comparison between the number of generations, which were required by the GSO, PSO and GA techniques, to generate test paths to cover all the def-uses pairs of each tested program.

Figure 9 shows a comparison between the number of test paths generated by the GSO, PSO and the GA techniques to cover all/possible def-uses. As can be seen from this figure, GSO generated less number of test paths than GA in 7 out of the 15 programs, and PSO in 3 out of the 15 programs. In 7 programs, the three techniques generated same number of test paths, and in 12 programs, both PSO and GSO generated same number of test paths. In program P#13, although GA generated less number of test paths than GSO, it did not reach 100% coverage as the GSO technique.

```
/* Hybrid GSO algorithm to automatically generate test paths that
cover the all-uses criterion for a given program */
Input:
    The program to be tested P;
    List of program def-use paths to be covered;
    List of edges and DD-graph of P;
    Population size;
    Maximum no. of generations (MAXGENS);
    c1, c2: learning factors;
    Probabilities of crossover and mutation;
Output:
    Set of test paths and set of def-use paths covered by each test
    path;
Begin
  Step 1: Initialization
      Initialize the def-use coverage vector to zeros;
      Randomly create Initial_Population of chromosomes (test
      paths) such that generated test paths satisfy the connectivity
      condition;
      GSO_Population ← initial population;
      def-use-coverage percent ← 0
      accumulated def-use-coverage percent ← 0
      No_of_Generations ← 0;
      nPaths ← 0;
      Calculate pBest for each member of GSO_Population;
  Step 2: Generate test paths
      While (Coverage_Percent ≠100 and
                    No_of_Generations ≤ MAXGENS) do
        For each member of GSO_Population do
          Convert GSO_Population chromosome to the
          corresponding path;
          Evaluate current test path (calculate fitness value);
          If (some def-use paths are covered) then
            nPaths ← nPaths + 1;
            Add effective test path to set of test paths for P;
            Update the def-use coverage vector;
            Update accumulated def-use-coverage percent;
          End If
        End For;
        Randomly generate hybridization coefficient (hc) such
        that hc ∈ [0, POPSIZE]
        If (0 < hc < POPSIZE) then
          Split GSO_Population into two parts according to hc:
          GA_Population and PSO_Population;
          Apply GA selection, crossover, mutation operations to
          GA_Population to create New_GA_Population;
          GA_Population ← New_GA_Population;
          Apply PSO position and velocity update operations to
          PSO_Population to create New_PSO_Population;
          PSO _Population ← New_ PSO _Population;
          GSO_Population ← GA_Population ∪
                                        PSO _Population;
        Else If (hc = 0) then
          PSO _Population ← GSO_Population
          Apply PSO position and velocity update operations to
          PSO_Population to create New_PSO_Population;
          PSO _Population ←New_ PSO _Population;
          GSO_Population ← PSO _Population;
        Else If (hc = POPSIZE) then
          GA_Population ← GSO_Population;
          Apply GA selection, crossover, mutation operations to
          GA_Population to create New_GA_Population;
          GA_Population ← New_GA_Population;
          GSO_Population ← GA_Population;
        End If
        Increment No_of_Generations;
      End While
  Step 3: Produce output
      Return set of test paths for P and set of def-use paths
      covered by each path;
End.
```

Figure 6: The proposed GSO-based test paths generation algorithm

```
Population Size: 6
Maximum Number of Generations: 100
CROSSOVER AND MUTATION PROBABILITES: 0.8, 0.15
Learning factors c1: 3          Learning factors c2: 3
** GSO Started **
* INITIAL POPOULATION *
1. 111011100100101  e0,e1,e2,e4,e6,e12,e9,e5,e14,    (Path 1)
2. 111011100000111  e0,e1,e2,e4,e6,e12,e13,e5,e14,   (Path 2)
3. 110111000110101  e0,e1,e3,e4,e10,e12,e9,e5,e14,   (Path 3)
4. 111011000000001  e0,e1,e2,e4,e5,e14,              (Path 4)
5. 110111001110001  e0,e1,e3,e4,e10,e8,e9,e5,e14,    (Path 5)
6. 110111001010011  e0,e1,e3,e4,e10,e8,e13,e5,e14,   (Path 6)
Path 1 ************** SELECTED **************
* Traversed Path: 5,6,7,8,9,10,14,15,16,17,18,19,20,21,22,23,24,
  29,30,19,31,32,
* DEF-USE COVERAGE:                          47.83 %
* ACCUMULATED DEF-USE COVERAGE:              47.83 %
* COVERED DEF-USE PATHS: 1,3,5,6,7,8,10,17,20,21,23
Path 3 ************** SELECTED **************
* Traversed Path:5,6,7,8,9,10,11,12,13,18,19,20,21,22,23,24,29,
  30,19,31,32,
* DEF-USE COVERAGE:                          13.04 %
* ACCUMULATED DEF-USE COVERAGE:              60.87 %
* COVERED DEF-USE PATHS: 2,4,9
Path 4: ************** SELECTED **************
* Traversed Path: 5,6,7,8,9,10,14,15,16,17,18,19,31,32,
* DEF-USE COVERAGE:                          4.35 %
* ACCUMULATED DEF-USE COVERAGE:              65.22 %
* COVERED DEF-USE PATHS: 22
* * hc = 3 **
** Apply GA to 3 individuals from the current population
* Parent 1 (Individual 5): 110111001110001e0,e1,e3,e4,e10,e8,e9,e5,e14,
* Parent 2 (Individual 3): 110111000110101 e0,e1,e3,e4,e10,e12,e9,e5,e14,
* Parent 3 (Individual 2): 111011100000111 e0,e1,e2,e4,e6,e12,e13,e5,e14,
Selected Cases to be Parents of New Population are:
* Parent 1 (Individual 2): 110111000110101 e0,e1,e3,e4,e10,e12,e9,e5,e14,
* Parent 2 (Individual 2): 110111000110101 e0,e1,e3,e4,e10,e12,e9,e5,e14,
* Parent 3 (Individual 2): 110111000110101 e0,e1,e3,e4,e10,e12,e9,e5,e14,
** Apply PSO to 3 individuals from the current population
* Parent 1 (Individual 1): 111011100100101
                    e0,e1,e2,e4,e6,e12,e9,e5,e14,
* Parent 2 (Individual 4): 111011000000001 e0,e1,e2,e4,e5,e14,
* Parent 3 (Individual 6): 110111001010011
                    e0,e1,e3,e4,e10,e8,e13,e5,e14,
particle    1       successfully updated as follows:
position before update 0,1,2,4,5,6,9,12,14,-1,-1,-1,-1,-1,-1,-1,
          binary vector:   111011100100101
velocity after update 0,0,8,14,14,12,14,10,13,2,8,13,14,12,15,
position after update 0,1,2,3,6,7,8,10,11,12,13,14,-1,-1,-1,
position after repair  0,1,2,4,6,7,9,10,8,13,5,14 = 111011111110011
particle    2       successfully updated as follows:
position before update  0,1,2,4,5,14,-1,-1,-1,-1,-1,-1,-1,-1,-1,
          binary vector:   111011000000001
velocity after update 0,0,12,6,7,10,6,9,8,13,14,2,12,4,15,
position after update 0,1,2,3,4,6,8,9,10,12,13,14,-1,-1,-1,
position after repair  0,1,2,4,6,8,9,10,12,13,5,14 = 111011101110111
particle    3       successfully updated as follows:
position before update  0,1,3,4,5,8,10,13,14,-1,-1,-1,-1,-1,-1,-1,
          binary vector:   110111001010011
velocity after update 0,0,14,11,14,10,9,14,8,10,14,10,14,10,15,
position after update 0,1,3,5,6,7,10,11,14,-1,-1,-1,-1,-1,-1,
position after repair 0,1,3,4,6,7,9,5,14 = 110111110100001
*** New Population:
1. 110111000110101  e0,e1,e3,e4,e10,e12,e9,e5,e14,          (Path 7)
2. 110111000110101  e0,e1,e3,e4,e10,e12,e9,e5,e14,          (Path 8)
3. 110111000110101  e0,e1,e3,e4,e10,e12,e9,e5,e14,          (Path 9)
4. 111011111110011  e0,e1,e2,e4,e6,e7,e9,e10,e8,e13,e5,e14, (Path 10)
5. 111011101110111  e0,e1,e2,e4,e6,e8,e9,e10,e12,e13,e5,e14, (Path 11)
6. 110111110100001  e0,e1,e3,e4,e6,e7,e9,e5,e14,           (Path 12)
Path 10 ************** SELECTED **************
* Traversed Path: 5,6,7,8,9,10,14,15,16,17,18,19,20,21,25,26,27,
  28,29,30,19,20,21,22,23, 24,29,30,19,31,32,
* DEF-USE COVERAGE:                          17.39 %
* ACCUMULATED DEF-USE COVERAGE:              82.61 %
* COVERED DEF-USE PATHS: 13, 14, 18, 19
Path 11 ************** SELECTED **************
* Traversed Path: 5,6,7,8,9,10,14,15,16,17,18,19,20,21,22,23,
  24,29,30,19,20,21,22,23,24,29,30,19,31,32,
* DEF-USE COVERAGE:                          4.35 %
* ACCUMULATED DEF-USE COVERAGE:              86.96 %
* COVERED Def-Use PATHS: 11
  Path 12 ************** SELECTED **************
  * Traversed Path: 5,6,7,8,9,10,11,12,13,18,19,20,21,25,26,27,28,29,
    30,19,31,32,
* DEF-USE COVERAGE:                          4.35 %
* ACCUMULATED DEF-USE COVERAGE:              91.30 %
* COVERED DEF-USE PATHS: 12
** hc= 5 **
** Apply GA to 5 individuals from the current population
* Parent 1 (Individual 1): 110111000110101
                    e0,e1,e3,e4,e10,e12,e9,e5,e14,
* Parent 2 (Individual 6): 110111110100001
                    e0,e1,e3,e4,e6,e7,e9,e5,e14,
* Parent 3 (Individual 5):  111011101110111
                    e0,e1,e2,e4,e6,e8,e9,e10,e12,e13,e5,e14,
* Parent  4 (Individual 4): 111011111110011
                    e0,e1,e2,e4,e6,e7,e9,e10,e8,e13,e5,e14,
* Parent 5 (Individual 3): 110111000110101
                    e0,e1,e3,e4,e10,e12,e9,e5,e14,
  * Selected Cases to be Parents of New Population are:
* Parent 1 (Individual 4): 111011111110011
                    e0,e1,e2,e4,e6,e7,e9,e10,e8,e13,e5,e14,
* Parent 2 ( Individual 2): 110111110100001
                    e0,e1,e3,e4,e6,e7,e9,e5,e14,
* Parent 3 (Individual 4): 111011111110011
                    e0,e1,e2,e4,e6,e7,e9,e10,e8,e13,e5,e14,
* Parent 4 (Individual 4): 111011111110011
                    e0,e1,e2,e4,e6,e7,e9,e10,e8,e13,e5,e14,
* Parent 5 (Individual 3): 111011101110111
                    e0,e1,e2,e4,e6,e8,e9,e10,e12,e13,e5,e14,
** Crossover
* Selected Parents: 1, 2          Crossover Position: 6
* Offspring:          110111111110011     111011110100001
** Mutation
* Selected Chromosome: 1          Mutation Position: 7          bits: 12
* Mutated Chromosome          110111101110111
** Apply PSO to 1 individuals from the current population
* Individual 2 (Parent  1):  110111000110101
                    e0,e1,e3,e4,e10,e12,e9,e5,e14,
particle    1       successfully updated as follows:
position before update  0,1,3,4,5,9,10,12,14,-1,-1,-1,-1,-1,-1,-1,
          binary vector:   110111000110101
velocity after update 0,0,9,13,13,2,5,8,10,10,4,14,9,8,15,
position after update 0,1,3,4,5,6,8,9,10,11,12,14,-1,-1,-1,
position after repair  0,1,3,4,6,8,9,10,11,13,5,14 = 110111101111011
*** New Population:
1. 110111101110111 e0,e1,e3,e4,e6,e8,e9,e10,e12,e13,e5,e14,  (Path 13)
2. 111011110100001  e0,e1,e2,e4,e6,e7,e9,e5,e14,             (Path 14)
3. 111011111110011 e0,e1,e2,e4,e6,e7,e9,e10,e8,e13,e5,e14,  (Path 15)
4. 111011111110011 e0,e1,e2,e4,e6,e7,e9,e10,e8,e13,e5,e14,  (Path 16)
5. 111011101110111  e0,e1,e2,e4,e6,e8,e9,e10,e12,e13,e5,e14, (Path 17)
6. 110111101111011 e0,e1,e3,e4,e6,e8,e9,e10,e11,e13,e5,e14,  (Path 18)
Path 18 ************** SELECTED **************
* Traversed Path: 5,6,7,8,9,10,11,12,13,18,19,20,21,22,23,24,
  29,30,19,20,21,25,26,27,28,29,30,19,31,32,
* DEF-USE COVERAGE:                          8.70 %
* ACCUMULATED DEF-USE COVERAGE:              100.00 %
* COVERED DEF-USE PATHS: 15, 16
** GSO TERMINATED **
** NO. OF GENERATIONS = 3
** GENERATED TEST PATHS **
111011100100101 e0,e1,e2,e4,e6,e12,e9,e5,e14,              (Path 1)
110111000110101 e0,e1,e3,e4,e10,e12,e9,e5,e14,             (Path 3)
111011000000001 e0,e1,e2,e4,e5,e14,                        (Path 4)
111011111110011 e0,e1,e2,e4,e6,e7,e9,e10,e8,e13,e5,e14,    (Path 10)
111011101110111 e0,e1,e2,e4,e6,e8,e9,e10,e12,e13,e5,e14,   (Path 11)
110111110100001 e0,e1,e3,e4,e6,e7,e9,e5,e14,               (Path 12)
110111101111011 e0,e1,e3,e4,e6,e8,e9,e10,e11,e13,e5,e14,   (Path 18)
```

Figure 7: Part of the report produced by the proposed GSO algorithm for the example program

Table 2: A comparison between the GSO, PSO and GA techniques

Prog#	No. of generations			No. of test paths			Def-use coverage %		
	GA	PSO	GSO	GA	PSO	GSO	GA	PSO	GSO
P#1	4	3	2	3	2	2	100	100	100
P#2	4	2	2	3	2	2	100	100	100
P#3	9	7	7	4	4	3	100	100	100
P#4	100	100	100	4	3	3	88.89	88.89	88.89
P#5	10	8	2	7	6	5	100	100	100
P#6	4	3	2	2	2	2	100	100	100
P#7	13	6	3	5	5	5	100	100	100
P#8	29	20	10	4	4	4	100	100	100
P#9	6	4	2	3	3	3	100	100	100
P#10	16	8	5	4	4	4	100	100	100
P#11	8	6	2	4	4	4	100	100	100
P#12	28	8	6	9	8	8	100	100	100
P#13	100	12	11	3	5	5	91.11	100	100
P#14	6	4	2	5	5	5	100	100	100
P#15	6	5	4	3	3	2	100	100	100

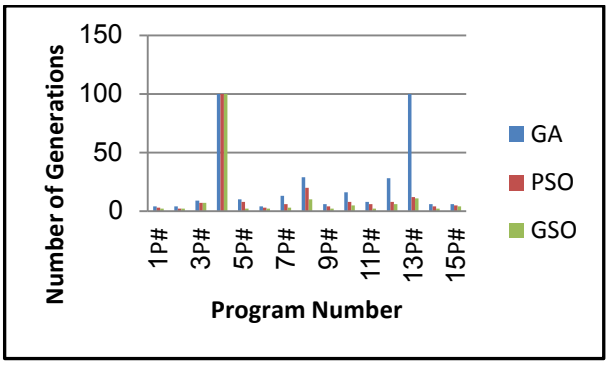

Figure 8: A comparison between the number of generations required by the GSO, PSO and GA techniques to generate test paths to cover all def-uses

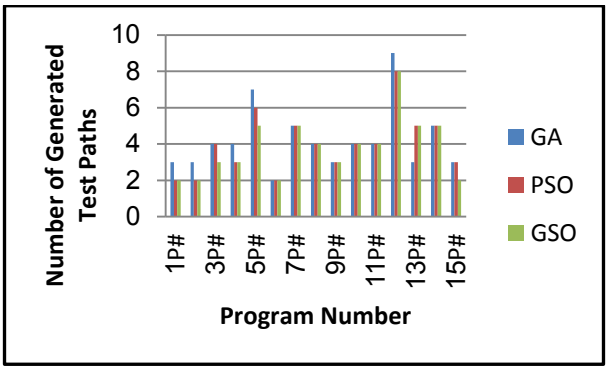

Figure 9: Comparing the No. of test paths generated by the GSO, PSO and GA techniques to cover all def-uses

7. CONCLUSION

This paper presented two proposed GA-based and PSO-based techniques for automatic generation of a set of test paths that cover the all-uses criterion for the program under test. These two techniques conduct their search by constructing new paths from previously generated paths that are evaluated as effective test paths. Then, the paper presented a GSO-based technique that effectively combines the proposed GA-based and the PSO-based techniques to improve the individuals score for natural selection of the fitness and for good-knowledge sharing at the same time. In each iteration of the proposed GSO algorithm, the population is divided into two parts and they are evolved with the two techniques respectively. They are then recombined in the updated population, that is again divided randomly into two parts in the next iteration for another run of genetic or particle swarm operators.

Experiments have been carried out to evaluate the effectiveness of the proposed GA, PSO and GSO approaches. The results of the experiments showed that the GSO technique required less number of generations to cover all def-use paths than the GA technique and the PSO technique. The results showed also that, in many cases, the GSO generated less number of test paths to cover all def-use paths than the GA and the PSO techniques. In general, the experiments indicated that the GSO approach is more effective in test paths generation than the GA and PSO approaches, and the PSO approach is more effective than the GA approach.

8. REFERENCES

[1] Bertolino, A., Marre, M. 1994. Automatic Generation of Path Covers Based on the Control flow analysis of computer Programs. IEEE Transaction on Software on software Engineering, Vol. 20, No. 12, pp.885-899.

[2] McCabe, T. and Thomas, J. 1982. Structural Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric, NIST Special Publication 500-99, Washington D.C.

[3] Poole, J. 2004. A Method to Determine a Basis Set of Paths to Perform Program Testing. http://hissa.nist.gov// publications/nistir5737.

[4] Guangmei, Z., Rui, C., Xiaowei, L., and Congying H. 2005. The Automatic Generation of Basis Set of Path for Path Testing. 14th Asian Test Symposium (ATS '05).

[5] Yan, J. and Zhang J. 2008. An efficient method to generate feasible paths for basis path testing. Information Processing Letters, Vol. 107, No. 3-4, pp. 87-92.

[6] Zhonglin, Z. and Lingxia, M. 2010. An Improved Method of Acquiring Basis Path for Software Testing. 5th International Conference on Computer Science & Education, pp.1891-1894, China.

[7] Qingfeng, D. and Xiao D. 2011. An Improved Algorithm for Basis Path Testing. International Conference on Business Management and Electronic Information (BMEI), pp. 175 – 178.

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

[9] Roper, M., Maclean, I., Brooks, A., Miller, J. and Wood, M. 1995. Genetic Algorithms and the Automatic Generation of Test Data. Technical Report RR/95/195 [EFoCS-19-95], University of Strathclyde, Glasgow G1 1XH, U.K.

[10] Watkins, A. E. L. 1995. A Tool for the Automatic Generation of Test Data Using Genetic Algorithms. Proceedings of Software Quality Conference, Dundee, Scotland.

[11] Jones, B. F. Eyres, D. E. and Sthamer, H. -H. 1998. A strategy for using genetic algorithms to automate branch and fault-based testing. The Computer Journal, Vol. 41, No. 2, pp. 98-107.

[12] Pargas, R. P., Harrold, M. J., and Peck, R. R. 1999. Test-Data Generation Using Genetic Algorithms. The Journal of Software Testing, Verification and Reliability.

[13] Lin J. -C. and Yeh, P. -L. 2001. Automatic test data generation for path testing using GAs. Information Sciences, Vol. 131, No. 1-4, pp. 47-64.

[14] Michael, C. C., McGraw, G. and Schatz, M. A. 2001. Generating Software Test Data by Evolution. IEEE Transactions on Software Engineering, Vol. 27, No. 12, pp. 1085-1110.

[15] Bueno P. M. S. and Jino, M. 2002. 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.

[16] Girgis, M. R. 2005. Automatic test data generation for data flow testing using a genetic algorithm. Journal of Universal computer Science, Vol. 11, No.5, pp. 898-915.

[17] Ghiduk, A. S., Harrold, M. J., Girgis, M. R. 2007. Using genetic algorithms to aid test-data generation for data flow coverage. 14th Asia-Pacific Software Engineering Conference (APSEC 07), pp. 41-48. IEEE Press.

[18] Gong, D. W., Zhang, W. Q. and Yao, X. J. 2011. Evolutionary Generation of Test Data for Many Paths Coverage Based on Grouping. Journal of Systems and Software, Vol. 84, No. 12, pp. 2222–2233.

[19] Girgis, M. R., Ghiduk, A. S. and Abd-Elkawy, E. H. 2013. An Approach For Enhancing Regression Testing Using Genetic Algorithm and Data Flow Analysis, International Journal of Intelligent Computing and Information Science, Vol. 13, No. 2, pp. 115-132.

[20] Hermadi, I., Lokan, C. and Sarker, R. 2010. Genetic Algorithm Based Path Testing: Challenges and Key Parameters. Second WRI World Congress on Software Engineering.

[21] Bint, J. R. and Site, R. 2004. Optimizing Testing Efficiency with Error Prone Path Identification and Genetic Algorithms. Australian Software Engineering Conference (ASWEC'04), Australia, pp. 106-115.

[22] Ghiduk, A. S., Said, O. and Aljahdali, S. 2012. Basis Test Paths Generation Using Genetic Algorithm. The 1st Taibah University International Conference on Computing and Information Technology (ICCIT 2012), pp. 303-308.

[23] Kennedy, J. and Eberhart, R. C. 1995. Particle Swarm Optimization. IEEE International Conference on Neural Networks, pp. 1942-1948.

[24] Li, A. G., Zhang, Y. L. 2008. Automatic generation method of test data for software structure based on PSO. Computer Engineering, Vol. 34, No. 6, pp. 93-97.

[25] Bueno, P. M. S., Wong, W. E. and Jino, M. 2008. Automatic test data generation using particle systems. ACM Symposium of Applied Computing pp. 809-814, Fortaleza, Brazil.

[26] Li, A. and Zhang, Y. 2009. Automatic generating all-path test data of a program based on PSO. The 2009 WRI World Congress on Software Engineering (WCSE'09), IEEE, Los Alamitos, Vol. 4, pp. 189-193.

[27] Agrawal, K. and Srivastava, G. 2010. Towards software test data generation using discrete quantum particle swarm optimization. ISEC, Mysore, India, pp. 65- 68.

[28] Narmada, N. and Mohapatra, D. P. 2010. Automatic Test Data Generation for data flow testing using Particle Swarm Optimization. Communications in Computer and Information Science, Vol. 95, No. 1, pp. 1-12.

[29] Nie, P., Geng, J. and Qin, Z. G. 2012. Multi-path oriented particle swarm optimization automatic test case generation algorithm. Computer Integrated Manufacturing Systems, Vol. 18, No. 1, pp. 216-223.

[30] Gandelli, A., Grimaccia, F., Mussetta, M., Pirinoli, P. and Zich, R. E. 2007. Development and Validation of Different Hybridization Strategies between GA and PSO. IEEE Congress on Evolutionary Computation (CEC 2007), pp. 2782-2787.

[31] Rapps S. and Weyuker, E. J. 1985. Selecting software test data using data flow information. IEEE Transactions on Software Engineering, Vol. 11, No. 4, pp. 367-375.

[32] Girgis, M. R. 1993. Using symbolic execution and data flow criteria to aid test data selection. The Journal of Software Testing, Verification and Reliability, Vol. 3, No. 2, pp. 101-112.

[33] Girgis, M. R. Ghiduk, A. S. Abd-Elkawy, E. H. 2014. Automatic Generation of Data Flow Test Paths using a Genetic Algorithm. International Journal of Computer Applications (0975 – 8887), Vol. 89, No. 12, pp. 29-36.

[34] Girgis, M. R. 1992. An experimental evaluation of a symbolic execution system. Software Engineering Journal, Vol. 7, No. 4, pp. 285-290.

[35] Goldberg, D. E. 1989. Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley, Reading, Mass.

[36] Eberhart, R. C., Shi, Y. 2000. Comparing inertia weights and constriction factors in particle swarm optimization. Proceedings of 2000 Congress on Evolutionary Computation, San Diego, CA, pp. 84–88.

[37] Clerc, M. 1999. The swarm and the queen: towards a deterministic and adaptive particle swarm optimization. 1999 Congress on Evolutionary Computation, Washington, DC, pp. 1951-1957. Piscataway, NJ, IEEE Service Center.