**Generating Test Data for Software Structural Testing using**

**Parallel Particle Swarm Optimization**

**Abstract**

Evolutionary structural testing is an approach to automatically generating test cases that achieve high structural code coverage. In recent investigations particle swarm optimization (PSO), an alternative search technique, often outperformed other meta-heuristic search techniques when applied to various problems. This paper proposes the approach of Parallel Particle Swarm Optimization (PPSO) in order to generate test data simultaneously for each test path of the given program under test (PUT). Experimental results demonstrate that PPSO which can generate suitable test data has higher path coverage than the previous one.

# 1. Introduction

Software products are becoming more and more important in today’s informatics society. Therefore, naturally, software quality has become the top concern in today’s informatics society. Software testing has proved itself to be one of the most efficient methods to assure and improve the software quality in the past few decades. However, as most of the software testing is being done manually, the workforce and cost required are accordingly high [1]. In general, about 50 percent of workforce and cost in the software development process is spent on software testing [2]. Considering those reasons, automated software testing has been evaluated as an efficient and necessary method in order to reduce those effort and costs.

In recent years, meta-heuristic search techniques has been widely applied in automatic software testing, forming a research trend called search-based software testing [3], which is especially applied to automatic test data generation. The general idea behind search-based test data generation is to select a set of test cases from program input space to meet the testing requirement which is usually expressed as a fitness function. When a coverage criterion is selected as the testing requirement, the search activity should attempt to generate a test suite which can cover all construct elements mentioned in the criterion.

Among the existing meta-heuristic search techniques, such as simulated annealing (SA) and generic algorithm (GA), are the most popular algorithms, and have been widely adopted in generating test data. However even though they can generate test data with appropriate fault-prone ability [4, 5], they fail to produce them quickly due to their slow evolutionary speed. Recently, as a swarm intelligence technique, particle swarm optimization (PSO) [6, 7, 8] has become a hot research topic in the area of intelligent computing. Its significant feature is the simplicity and fast convergence speed.

Even so, there are still certain limitations in current research on PSO usage in test data generation. For example, consider one program under test which was used in Mao’s paper [9] as below:

int getDayNum(int year, int month)

{

int maxDay=0;

if(month≥1 && month≤12) //bch1: branch 1

{

if(month=2) //bch2: branch 2

{

if(year%400=0||(year%4=0&&year%100=0))

//bch3: branch 3

maxDay=29;

else //bch4: branch 4

maxDay=28;

}

else if(month=4||month=6||month=9||month=11)

//bch5: branch 5

maxDay=30;

else //bch6: branch 6

maxDay=31;

}

else //bch7: branch 7

maxDay=-1;

return maxDay;

}

Regarding this program under test, Mao [9] used PSO to generate test data through building the one and only fitness function which was the combination of Korel formula [10] and the branch weights. This proposal has two weaknesses: the branch weight function being entirely performed manually and some PUTs not being able to generate test data to cover all test paths. To overcome these weaknesses, we still use PSO to generate test data for the given PUT. However, unlike Mao, our approach is to assign one fitness function for each test path. Then we will use PPSO to find simultaneously the solution corresponding to this fitness function, which is also the one being able to generate test data for this test path.

# 2. Background

## 2.1. Fitness function

When using PSO, a test path coverage test data generation is transformed into an optimization problem. To cover a test path during execution, we must find appropriate values for the input variables which satisfy related branch predicates. The usual way is to use Korel’s branch distance function [10]. As a result, generating test data for a desired branch is transformed into searching input values which minimizes the return value of its Korel function. Table 1 gives some common formulas which are used in branch distance functions. To generate test data for a desired path P, we define a fitness function *F*(*P*) as the sum of all related branch distance functions. For these reasons, generating path coverage test data can be converted into searching input values which can minimize the return value of function *F*(*P*).

**Table 1.** Korel’s branch functions for several kinds of branch predicates

|  |  |  |
| --- | --- | --- |
| No | Relational predicate | Branch distance function *f*(bch*i*) |
| 1 | Boolean | If *true* then 0 else *k* |
| 2 | ¬*a* | Negation is propagated over *a* |
| 3 | *a* = *b* | If abs(*a* – *b*)= 0 then 0 else abs(*a* − *b*)+ *k* |
| 4 | *a* ≠ *b* | If abs(*a* − *b*)≠ 0 then 0 else *k* |
| 5 | *a* < *b* | If *a* − *b <* 0 then 0 else abs(*a* − *b*)+ *k* |
| 6 | *a* ≤ *b* | If *a* − *b* ≤ 0 then 0 else abs(*a* − *b*)+ *k* |
| 7 | *a* > *b* | If *b* − *a >* 0 then 0 else abs(*b* − *a*)+ *k* |
| 8 | *a* ≥ *b* | If *b* − *a* ≥ 0 then 0 else abs(*b* − *a*)+ *k* |
| 9 | *a* and *b* | *f* (*a*)+ *f*(*b*) |
| 10 | *a* or *b* | min(*f*(*a*)*, f*(*b*)) |

Similar to Mao [9], we also set up the value k = 0.1. Basing on this formula, we will develop a function calculating values at branch predication, which is to be explained in the next part.

## 2.2. Particle Swarm Optimization

Particle Swarm Optimization (PSO) was first introduced in 1995 by Kennedy and Eberhart [11], and is now widely applied in optimization problems. Comparing to other optimal search algorithms such as GA or SA, PSO has the strength of faster convergent speed and easier coding. PSO is initialized with a group of random particles (solutions) and then it searches for optima by updating generations. In every iteration, each particle is updated by the following two "best" values. The first one is the best solution (fitness) it has achieved so far (the fitness value is also stored). This value is called *pbest*. Another "best" value that is tracked by the particle swarm optimizer is the best value, obtained so far by any particle in the population. This best value is a global best and called *gbest*.

After finding the two best values, the particle updates its velocity and positions with following equation (a) and (b).  
(a)

(b)

*v*[] is the particle velocity, *persent*[] is the current particle (solution). *pbest*[] and *gbest*[] are defined as stated before. *rand*() is a random number between (0,1). *c*1, *c*2 are learning factors, usually *c*1 = *c*2 = 2.

The PSO algorithm is described by pseudo code as below:

|  |
| --- |
| **Algorithm 1**: Particle Swarm Optimization (PSO) |
| **Input:** *F*: Fitness function |
| **Output:** *gBest*: The best solution |
| 1: **for each** particle |
| 2:    initialize particle |
| 3: **end for** |
| 4: **do** |
| 5: **for each** particle |
| 6: calculate fitness value |
| 7: **if** the fitness value is better than the best fitness value (*pBest*) in history then |
| 8:   set current value as the new *pBest* |
| 9: **end if** |
| 10: **end for** |
| 11: choose the particle with the best fitness value of all the particles as the *gBest* |
| 12: **for each** particle |
| 13: calculate particle velocity according equation (a) |
| 14:   update particle position according equation (b) |
| 15: **end for** |
| 16: **while** maximum iterations or minimum criteria is not attained |

Particles' velocities on each dimension are clamped to a maximum velocity *Vmax*, which is an input parameter specified by the user.

# 3. Related work

PSO algorithm, which simulates to birds flocking around food sources, was invented by Kennedy and Eberhart [11] in 1995, and was originally just an algorithm used for optimization problems. However with the advantages of the convergence speed and easier construction than other optimization algorithms, it was promptly adopted as a meta-heuristic search algorithm in the automatic test data generation problem.

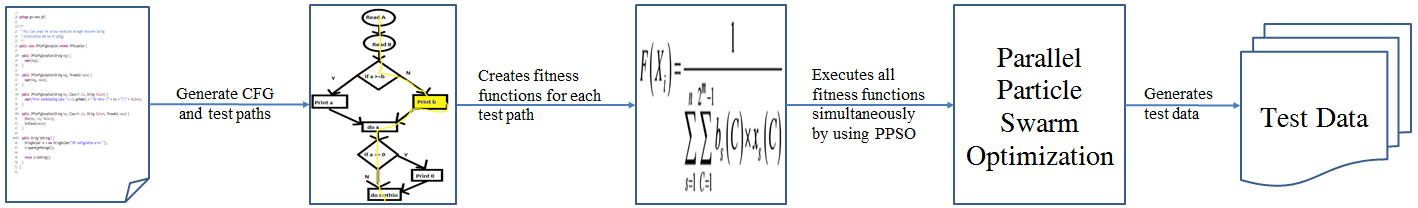
Windisch et al. [6] were the first authors to apply PSO in automatic test data generation. They improved the PSO into comprehensive learning particle swarm optimization (CL-PSO) to generate structural test data, but some experiments proved that the convergence speed of CL-PSO was perhaps worse than the basic PSO.

Jia et al. [7] created an automatic test data generating tool named particle swarm optimization data generation tool (PSODGT). The PSODGT is characterized by two features. First, the PSODGT adopts the condition-decision coverage as the criterion of software testing, aiming to build an efficient test data set that covers all conditions. Second, the PSODGT uses a particle swarm optimization (PSO) approach to generate test data set. In addition, a new position initialization technique is developed for PSO. Instead of initializing the test data randomly, the proposed technique uses the previously-found test data which can reach the target condition as the initial positions so that the search speed of PSODGT can be further accelerated. The PSODGT is tested on four practical programs.

Mao [9] and Zhang et al. [8] had the same approach, in which they did not execute any PSO improvement but only built a fitness function by combining the branch functions for branch predicates and the branch weight of a program under test, then applied PSO to find the solution for this fitness function. The experiment result with 1 benchmark having 8 programs under test proved that PSO algorithm was more effective than GA in generating test data. However, there remained a weakness that the calculation of branch weight for a program under test was still entirely manual work, which reduced the automatic nature of the proposal. In this paper, our proposal can overcome this limitation while being able to assure the efficiency of a PSO-based automatic test data generation method.

# 4. Proposed approach

Our proposed approach can be divided into two separate parts: performing statistical analysis and applying PPSO to generate test data. It is presented as below chart:



**Fig. 1.** The basic steps for PPSO-based test data generation

## 4.1. Perform statistical analysis to find out all test paths

At first, we perform the statistical analysis to find all test paths of the program under test. It can be done through the below 2 small steps:

*1) Control flow graph generation:* Test case generated from source code directly is more complicated and difficult than from control flow graph (CFG). CFG is a directed graph visualizing logic structures of program simplify [12] and is defined as follow:

**Definition 1 (CFG).** *Given a function, a corresponding CFG is defined as a pair G* =(*V*, *E*), *where V* ={*v*0*, v1*,…*vn*} *is a set of vertices representing statements, E =* {(*vi, vj*)*|vi, vj V*}⊂ *V V is a set of edges. Each edge* (*vi*, *vj*) *implies the statement corresponding to vj is executed after vi.*

|  |
| --- |
| **Algorithm 2**: GenerateCFG |
| **Input** : *f* : source code |
| **Output**: *graph*: CFG |
| 1: *B* = a set of blocks by dividing *f* |
| 2: *G* = a graph by linking all blocks in *B* to each other |
| 3: update *graph* by replacing *f* with *G* |
| 4: **if** *G* contains *return/break/continue* statements **then** |
| 5: update the destination of *return/break/continue* pointers in the *graph* |
| 6: **end if** |
| 7: **for** each block *M* in *B* do |
| 8: **if** block *M* can be divided into smaller blocks **then** |
| 9: GenerateCFG(*M*) |
| 10: **end if** |
| 11: **end for** |

Apply this GenerateCFG algorithm to the above mentioned PUT getDayNum, we will get a CFG as below chart:



**Fig. 2.** CFG of PUT getDayNum

*2) Test paths generation:*

In order to generate test data, a set of feasible test paths is discovered by traversing the given CFG. Path and test path are defined as follows:

**Definition 2 (Path).** *Given a CFG G =* (*V, E*)*, a path is a sequence of vertices* {*v0, v*1*,..., vk |* (*vi, vi*+1) *E,* 0< *k* < *n*}*, where n is the number of vertices.*

**Definition 3 (Test path).** *Given a CFG G =* (*V, E*)*, a test path is a path* {*v*0*, v*1*,..., vk |* (*vi, vi*+1) *E*}*, where v*0 *and vi+*1 *are corresponding to the start vertex and end vertex of the CFG.*

|  |
| --- |
| **Algorithm 3**: TraverseCFG |
| **Input** : *v*: the initial vertex of the CFG  *depth*: the maximum number of iterations for a loop  *path*: a global variable used to store a discovered test path |
| **Output**: *P*: a set of feasible test paths |
| 1: **if** *v* = NULL or *v* is the end vertex then |
| 2: add *path* to *P* |
| 3: **else if** the number occurrences of *v* in *path* ≤ *depth* **then** |
| 4: add *v* to the end of *path* |
| 5: **if** (*v* is not a decision node) **or** (*v* is decision node and *path* is feasible) **then** |
| 6: **for each** adjacent vertex *u* to *v* **do** |
| 7: TraverseCFG(*u*, *depth*, *path*) |
| 8: **end for** |
| 9: **end if** |
| 10: remove the latest vertex added in *path* from it |
| 11: **end if** |

## In this paper, a test path is represented as a sequence of pairs of predicate, e.g. (*month* ≥ 1 && *month* ≤ 12) for the first branch, and its decision (T or F for TRUE or FALSE respectively). For example, one of the paths in PUT getDayNum can be written as the sequence {[(*month* ≥ 1 && *month* ≤ 12), T], [(*month* = 2), T], [(*year* % 400 = 0 || (*year* % 4 = 0 && *year* % 100 = 0)), F]} which means the TRUE branch is taken at predicate (*month* ≥ 1 && *month* ≤ 12), the TRUE branch at predicate (*month* = 2), and the FALSE branch at predicate (*year* % 400 = 0 || (*year* % 4 = 0 && *year* % 100 = 0)). This is the path taken with data that represents the number of days of February of the not leap year. Apply this algorithm TraverseCFG to the CFG of PUT getDayNum, we will get 5 test paths which are presented as a sequence of pairs of branch predication and its decisions as in the Table 2 below:

**Table 2.** All test paths of PUT getDayNum

|  |  |  |
| --- | --- | --- |
| No | PathID | Path’s branch predications and their decisions |
| 1 | path1 | [(*month* ≥ 1 && *month* ≤ 12), T], [(*month* = 2), T],  [(*year* % 400 = 0 | | (*year* % 4 = 0 && *year* % 100 = 0)), T] |
| 2 | path2 | [(*month* ≥ 1 && *month* ≤ 12), T], [(*month* = 2), T], [(*year* % 400 = 0 || (*year* % 4 = 0 && *year* % 100 = 0)), F] |
| 3 | path3 | [(*month* ≥ 1 && *month* ≤ 12), T], [(*month* = 2), F], [(*month* = 4|| *month* = 6|| *month* = 9 || *month* = 11), T] |
| 4 | path4 | [(*month* ≥ 1 && *month* ≤ 12), T], [(*month* =2), F], [(*month* = 4|| *month* = 6|| *month* = 9 || *month* =11), F] |
| 5 | path5 | [(*month* ≥ 1 && *month* ≤ 12), F] |

## 4.2. Establish fitness function for each test path

From the branch distance calculation formula in Table 1, we develop the below function *fBchDist* to calculate the value at each predicate branch.

|  |
| --- |
| **Algorithm 4**: Branch distance function (*fBchDist*) |
| **Input:** double a, condition type, double b |
| **Output:** Branch distance value |
| 1: **switch** (condition type)  2: **case** “=”: |
| 3: if abs(*a* − *b*) = 0 then retrun 0 else return abs(*a* − *b*) + *k*) |
| 4: **case** “≠”: |
| 5: if abs*(a* − *b)* ≠ 0 then return 0 else return *k* |
| 6: **case** “<”: |
| 7: if *a* − *b <* 0 then return 0 else return (abs*(a* − *b)* + *k*) |
| 8: **case** “≤”: |
| 9: if *a* − *b* ≤ 0 then return 0 else return (abs*(a* − *b)* + *k*) |
| 10: **case** “>”: |
| 11: if *b* − *a >* 0 then return 0 else return (abs*(b* − *a)* + *k*) |
| 12: **case** “≥”: |
| 13 if *b* − *a* ≥ 0 then return 0 else return (abs*(b* − *a)* + *k*) |
| 14: **end switch** |

Since each test path is represented by sequence of pairs of branch predication and its decision, in order to build the fitness function for the test path, we establish the fitness function for each branch predication and its decision. There will be 2 possibilities of TRUE(T) and FALSE(F) for each branch predication, so there will be 2 fitness functions corresponding to those possibilities. Regarding the calculation formula for the fitness function of each branch predication, we apply the above mentioned branch distance calculation algorithm.

**Table 3.** Fitness functions for branch predication and its decision of PUT getDayNum

|  |  |  |  |
| --- | --- | --- | --- |
| No | Decision node | Fitness function | ID |
| 1 | [(*month* ≥ 1 && *month* ≤ 12), T] | *fBchDist*(*month*, ≥, 1) + *fBchDist* (*month*, ≤, 12) | *f*1*T* |
| 2 | [(*month* ≥ 1 && *month* ≥ 12), F] | min(*fBchDist* (*month*, <, 1), *fBchDist* (*month*, >, 12)) | *f*1*F* |
| 3 | [(*month* = 2), T] | *fBchDist* (*month*, =, 2) | *f*2*T* |
| 4 | [(*month* = 2), F] | *fBchDist* (*month*, ≠, 2) | *f*2*F* |
| 5 | [(*year* % 400 = 0 ||  (*year* % 4 = 0 && *year* % 100 = 0)), T] | min(*fBchDist* (*year* %400, =, 0),  (*fBchDist* (*year* %4, =, 0) + *fBchDist* (*year* %100, =, 0))) | *f*3*T* |
| 6 | [(*year* %400 = 0 ||  (*year* % 4 = 0 && *year* % 100 = 0)), F] | *fBchDist* (*year* %400, ≠, 0) + min(*fBchDist* (*year* %4, ≠, 0), *fBchDist* (*year* %100, ≠, 0)) | *f*3*F* |
| 7 | [(*month* = 4 || *month* = 6 ||  *month* = 9 || *month* = 11), T] | min(*fBchDist* (*month*, =, 4), *fBchDist* (*month*, =, 6), *fBchDist* (*month*, =, 9), *fBchDist* (*month*, =, 11)) | *f*4*T* |
| 8 | [(*month* = 4 || *month* = 6 ||  *month* = 9 || *month* = 11), F] | *fBchDist* (*month*, ≠, 4) + *fBchDist* (*month*, ≠, 6) +  *fBchDist* (*month*, ≠, 9) + *fBchDist* (*month*, ≠, 11) | *f*4*F* |

From theses formulas to calculate fitness value for each branch predication, we generate the fitness function for each test path as below:

**Table 4.** Fitness functions for each test path of PUT getDayNum

|  |  |  |
| --- | --- | --- |
| No | PathID | Test path fitness functions |
| 1 | path1 | *F*1 = *f*1*T* + *f*2*T* + *f*3*T* |
| 2 | path2 | *F*2 = *f*1*T* + *f*2*T* + *f*3*F* |
| 3 | path3 | *F*3 = *f*1*T* + *f*2*F* + *f*4*T* |
| 4 | path4 | *F*4 = *f*1*T* + *f*2*F* + *f*4*F* |
| 5 | path5 | *F*5 = *f*1*F* |

## 4.3. Apply Parallel Particle Swarm Optimization

With each fitness function of each test path, we use one PSO to find its solution (in this case the solution means the test data which can cover the corresponding test path). In order to find the solution for all fitness functions at the same time, we perform parallelization of the PSO (PPSO) algorithm by defining PPSO it as 1 class extends Thread class of Java as follows:

public class PSOProcess extends Thread {}

The PSO parallelization can be executed through below algorithm:

|  |
| --- |
| **Algorithm 5**: Parallel Particle Swarm Optimization(PPSO) |
| **Input:** list of fitness function |
| **Output:** test data for each fitness function |
| 1: **for each** fitness function *Fi*  2: Initialize an object *psoi* of class PSOProcess |
| 3: Assign a fitness function *Fi* to object *psoi* |
| 4: Execute object *pso*: *pso.start()*; |
| 5: **end for** |

The experimental results of the above steps gave the results that our proposal has generated test data which covered all test paths of PUT getDayNum:

# 

# 5. Experimental analysis

We compare our experimental result to Mao’s proposal [9] in 2 criteria: the automatic ability of test data generation and the coverage capabilities of each proposal for each program under test of the given benchmark.

## 5.1. Automatic ability

When referring to an automatic test data generation method, the actual coverage of "automatic" ability is one of the key criteria to decide the proposal’s effectiveness. Mao [9] used only 1 fitness to generate test data for all test paths of a PUT, therefore he had to combine branch weight for each test path into the fitness function. The build of a branch weight function is purely manual, and for long and complex PUT, sometimes it is even harder than generating test data for the test paths, therefore it affected the efficiency of his proposed approach.

On the opposite site, taking advantage of the fast convergence of PSO algorithm, we propose the solution of using separate fitness function for each test path. This solution has clear benefits as follows:

1. As there is no need to build the branch weight function, the automatic feature of this proposal will be improved.
2. The fitness functions are automatically built basing on the pair of branch predication and its decision of each test path, and these pairs can be entirely generated automatically from a PUT with above mentioned algorithm 2 and 3. This obviously advanced the automatic ability in our proposal.

## 5.2. Coverage ability

We also confirmed our proposed approach on the benchmark which is used in Mao’s paper [9]. We compared the coverage ability to the 4 PUT in the benchmark that did not work well.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| PUT name | Lines of code | Branches | Arguments | Description |
| triangleType | 31 | 5 | 3 | Type classification for a triangle |
| computeTax | 61 | 11 | 2 | Compute the federal personal income tax |
| printCalendar | 187 | 33 | 2 | Print the calendar of a month in some year |
| line | 92 | 36 | 8 | Check if two rectangles overlap |

The two criteria to be compared with Mao’s result [9] are:

* Success rate (SR): the probability of all branches which can be covered by the generated test data. In order to check the actual result basing on this criterion, we executed PPSO by 1000 times, and calculated the number of times at which generated test data could cover all test paths of given PUT. The SR formula is calculated as follows:
* Average coverage (AC): the average of the branch coverage achieved by all test inputs in 1,000 runs. Similar to above, in order to check the actual result basing on this criterion, we executed PPSO by 1000 times, and calculated the average coverage for each run. AC formula is calculated for each PUT as follows:

The detail results of the comparison with PUT benchmark used by Mao [9] in 2 criteria are shown in the following table:

**Table 5.** Comparison between Mao's approach and PPSO

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Program under test | Success rate (%) | | Average coverage (%) | |
| Mao[10]’s PSO | PPSO | Mao[10]’s PSO | PPSO |
| triangleType | 99.80 | 100.0 | 99.94 | 100.0 |
| computeTax | 99.80 | 100.0 | 99.98 | 100.0 |
| printCalendar | 99.10 | 100.0 | 99.72 | 100.0 |
| line | 99.20 | 100.0 | 99.86 | 100.0 |

From Table 5 it can be see that there are some PUTs which Mao's proposed approach cannot fully cover, while our method can. Because each test path is assigned to a PSO, it ensures that every time the PPSO is run, each PSO can generate test data which can cover the test path it is assigned to.

# 6. Conclusion

This paper has introduced and evaluated a parallel PSO approach for the evolutionary structural testing. We proposed a method which uses a fitness function for each test path of a PUT, and then execute those PSOs simultaneously in order to generate test data to cover test paths of a PUT. The experimental result proves that our proposal is more effective than Mao’s [1] test data generation method using PSO in terms of both automatic and coverage ability for a PUT.

In future work, some issues should be incorporated into deep investigation. The search capability of PSO algorithm could be enhanced through absorbing some other strategies in intelligent computing. To exploit more reasonable form of fitness function is also a valuable research topic. At present, we only display the results of some benchmark programs from academe. So the experiments on some industrial programs are worthy of being deeply studied.

**References**

1. B. Antonia: Software Testing Research: Achievements, Challenges, Dreams. In: Future of Software Engineering, pp. 85--103. IEEE Computer Society, Washington (2007)
2. G. J. Myers: The Art of Software Testing, 2nd edition. John Wiley & Sons Inc (2004)
3. McMinn, P.: Search-based software testing: past, present and future. In: Proceedings of ICSE Workshop on the Search-Based Software Testing (SBST’11), pp. 153–163 (2011)
4. M. A. Ahmed and I. Hermadi: GA-based Multiple Paths Test Data Generator. Computers & Operations Research, vol. 35, pp 3107--3124 (2008).
5. J. Malburg and G. Fraser: Search-based testing using constraint-based mutation. Journal Software Testing, Verification & Reliability, vol. 24(6), 472--495 (2014).
6. Windisch, A.; Wappler, S.; Wegener, J.: Applying particle swarm optimization to software testing. In: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO’07), pp. 1121–1128 (2007)
7. Yanli Zhang, Aiguo Li, "Automatic Generating All-Path Test Data of a Program Based on PSO", vol. 04, pp. 189-193, 2009, doi:10.1109/WCSE.2009.98
8. Ya-Hui Jia, Wei-Neng Chen, Jun Zhang, Jing-Jing Li, Generating Software Test Data by Particle Swarm Optimization, in the Proceedings of 10th International Conference, SEAL 2014, Dunedin, New Zealand, December 15-18, 2014
9. C. Mao: Generating Test Data for Software Structural Testing Based on Particle Swarm Optimization. Arabian Journal for Science and Engineering, vol 39, issue 6, pp 4593–4607 (June 2014).
10. B. Korel. Automated software test data generation. IEEE Transactions on Software Engineering, vol. 16, 870-879 (1990).
11. Kennedy, J.; Eberhart, R.C.: Particle swam optimization. In: Proceedings of IEEE International Conference on Neural Networks (ICNN’95), pp. 1942–1948 (1995)
12. Robert Gold, Control flow graph and code coverage, in: Int. J. Appl. Math. Comput. Sci., Vol. 20, No. 4, 2010, pp. 739-749