VNU Journal of Science: Comp. Science & Com. Eng., Vol. 33, No. 2 (2017) 1-11

Generating Test Data for Software Structural Testing using Particle Swarm Optimization

Dinh Ngoc Thi*

VNU University of Engineering and Technology, 144 Xuan Thuy, Cau Giay, Hanoi, Vietnam

Abstract

Search-based test data generation is a very popular domain in the field of automatic test data generation. However, existing search-based test data generators suffer fromsome problems. By combining static program analysis and search-based testing, our proposed approach overcomesone of these problems. Considering the automatic ability and the path coverage as the test adequacycriterion, this paper proposes using Particle Swarm Optimization, an alternative search technique, for automating the generation of test data for evolutionary structural testing. Experimental results demonstrate that our test data generator can generate suitable test data with higher path coverage than the previous one.

Received June 2017; Revised November 2017; Accepted December 2017

Keywords: Automatic test data generation, search-based software testing, Particle Swarm Optimization.

1. Introduction

Software is amandatory part of today's life, and has become more and more important in current information society. However, its failure may lead to significanteconomic loss or threat to life safety. As a consequence, software qualityhas become a top concern today. Among the methods of software quality assurance, software testing has been proven as one of the effective approachesto ensure and improve software quality over the past threedecades. 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 efforts and costs.

Automated structural test data generation is becoming the research topic attracting much interest in automated software testingbecause it enhances the efficiency while reducing considerably costs of software testing. In our paper, we will focus on path coverage test data generation, considering that almost all structural test data generation problems can be transformed to the path coverage test datageneration one. Moreover, Kernighan and Plauger [3] also pointed out that path coverage test data generation can find out more than 65 percent of bugs in the given program under test (PUT).

Although path coverage test data generation is the major unsolved problem [20], various approaches have been proposed by researchers. These approaches can be classified into two types: constraint-based test data generation (CBTDG) or search-based test data generation (SBTDG).

* E-mail.: dinhngocthi@gmail.com
https://doi.org/10.25073/2588-1086/vnucsce.165

Symbolic execution (SE) is the state-of-the-art of CBTDG approaches [21]. Even though there have been significant achievements, SE still faces difficulties in handling infinite loops, array, procedure calls and pointer references in each PUT [22].

There are also random testing, local search [10], and evolutionary methods [23, 24, 25] in SBTDG approaches. As the value of input variables is assigned when a program executes, problems encountered in CBTDG approaches can be avoided in SBTDG.

Being an automated searching method in a predefined space, genetic algorithm (GA) was applied to test data generation since 1992 [26]. Micheal et al [22], Levin and Yehudai [25], Joachim et al [27] indicated that GA outperforms other SBTDG methods e.g. local search or random testing.However eventhough they can generate test data with appropriate fault-prone ability [4, 5], they fail to produce them quickly due to their slowly 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 its 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 PUT 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 PUT, 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 is entirely performed manually and some PUTs are not 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 simultaneous multithreading of PSO to simultaneously find the solution corresponding to this fitness function, which is also the one able to generate test data for this test path.

The rest of this paper is organized as follows: Section 2 gives some theoretical backgroundon fitness function and particle swarm optimization algorithm. Section 3 summarizes some related works, and Section 4 presents the proposed approach in detail. Section 5 shows the experimental results and discussions. Section 6 concludes the paper.

2. Background

This section describes the theoretical background being used in our proposed approach.

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 optimizes 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 total values 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 severalkinds of branch predicates

Relational predicate	Branch distance function $f(\text{bch}_i)$
Boolean	if *true* then 0 else k
$\neg a$	negation is propagated over a
$a = b$	if abs($a - b$)= 0 then 0 else abs($a - b$)+ k
$a \neq b$	if abs($a - b$)≠0 then 0 else k
$a<b$	if $a - b < 0$ then 0 else abs($a - b$)+ k
$a \leq b$	if $a - b \leq 0$ then 0 else abs($a - b$)+ k
$a>b$	if $b - a > 0$ then 0 else abs($b - a$)+ k
$a \geq b$	if $b - a \geq 0$ then 0 else abs($b - a$)+ k
a and b	$f(a)+ f(b)$
a or b	min($f(a)$, $f(b)$)

Similar to Mao [9], we also set up the punishment factor k = 0.1. Basing on this formula, we will develop a function calculating values at branch predication, which is will 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. Compared 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 (initial 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) achieved so far (the fitness value is also stored). This value is called *pbest*. Another "*best*" value 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 the following equation (1) and (2).

$$v[] = v[] + c1 \times rand() \times (pbest[] - present[]) + c2 \times rand() \times (gbest[] - present[]) \quad (1)$$

$$present[] = present[] + v[] \quad (2)$$

$v[]$ is the particle velocity, $persent[]$ is the current particle (currentsolution). $pbest[]$ and $gbest[]$ are defined as stated before. $rand()$ is a random number between (0,1). $c1$, $c2$ are learning factors, usually $c1 = c2 = 2$.

The PSO algorithm is described by pseudo code as shown 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 (1)
14: update particle position according equation (2)
15: **end for**
16: **while** maximum iterations or minimum criteria is not attained

Particles' velocities on each dimension are clamped to a maximum velocity V_{max}, which is aninput parameter specified by the user.

3. Related work

From the 1990s, genetic algorithm (GA) has been adopted to generate test data. Jones et. al. [13] presented a GA-based branch coverage test data generator. Their fitness function made use of weighted Hamming distance tobranch predicate values. They used unrolled control flow graph of a test program such that it is acyclic. Six small programs were used to test the approach.In recent years, Harman and McMinn [14] performed empirical study on GA-based test data generation for large-scale programs, and validated its effectiveness over other meta-heuristic search algorithms.

Although GA is a classical search algorithm, its convergence speed is not very significant. 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 faster convergence speed and easier constructionthan other optimization algorithms, it was promptly adopted as a meta-heuristic search algorithm in the automatic test data generation problem.

Automatic test data generation literature using PSO started with Windisch et al. [6] in 2007. They improved the PSO intocomprehensive 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. [8] 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.

Khushboo et al. [15] described the application of the discrete quantum particleswarm optimization (QPSO) to the problem of automated test data generation.Thediscrete quantum particle swarm optimization algorithm is proposed on the basis of the conceptof quantum computing. They had studied the role of the critical QPSO parameters on test data generation performance and based on observationsan adaptive version (AQPSO) had been designed. Its performance comparedwith QPSO. They used the branch coverage as their test adequacy criteria.

Tiwari et al. [16] had applied a variant of PSO in the creation of new test data formodified code in regression testing. The experimental resultsdemonstrated that this method could cover more code in lessnumber of iterations than the original PSO algorithm.

Zhu et al. [17] put forward an improved algorithm (APSO) and applied it to automatictest data generation, in which inertia weight was adjusted accordingto the particle fitness. The results showed that APSO had betterperformance than basic PSO.

Dahiya et al. [18] proposed a PSO-basedhybrid testing technique and solved many of the structural testingproblems such as dynamic variables, input dependent array index,abstract function calls, infeasible paths and loop handling.

Singla et al. [19] presented a technique on the basis of a combination ofgenetic algorithm and particle swarm algorithm. It is used togenerate automatic test data for data flow coverage by usingdominance concept between two nodes, which is compared toboth GA and

PSO for generation of automatic test cases todemonstrate its superiority.

Mao [9] and Zhang et al. [7] had the same approach, in which they did not execute any PSO improvement but only built a fitness function by combining the branch distance functions for branch predicates and the branch weights of a PUT, 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 PUT 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 static analysis and applying simultaneous multithreading of PSO to generate test data. This approach is presented in the Figure 1 below.

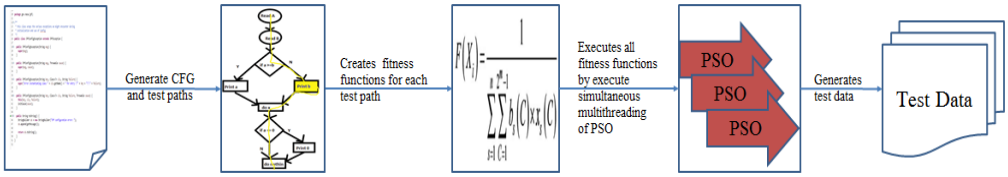

Figure 1. The basic steps for PSO-based test data generation.

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 given PUT. We call static analysis because of not having to execute the program, we can still generate control flowgraph (CFG) from the given program, and then traverse this CFG to find out all test paths.It can be done through the following two small steps:

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

Definition1(CFG). Given a program, a corresponding CFG is defined as a pair G =(V, E), where V ={v_0, v_1,…v_n} is a set of vertices representing statements, E ={(v_i, v_j)|v_i, v_j∈ V}⊂ V× V is a set of edges. Each edge (v_i, v_j) implies the statement corresponding to v_j is executed after v_i.

This paper uses the CFG generation algorithm from a given program which was presented in [28].Before performing this algorithm, output graph is initialized as a global variable and contains only one vertex representing for the given program P.

Algorithm 2: GenerateCFG
Input : *P* : given program
Output: *graph*: CFG
1: *B* = a set of blocks by dividing *P*
2: *G* = a graph by linking all blocks in *B* to each other
3: update *graph* by replacing *P* 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 which has 5 test paths (presented by decision nodes) as Figure 2 following.

2) *Test paths generation*:In order to generate test data, a set of feasible test paths is found 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 {v_0, v_1,..., v_k |(v_i, v_{i+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,..., v_k |(v_i, v_{i+1})∈ E}, where v_0 and v_{i+1} are corresponding to the start vertex and end vertex of the CFG.

This research also uses CFG traverse algorithm [28] to obtain feasible test paths from a CFG as below:

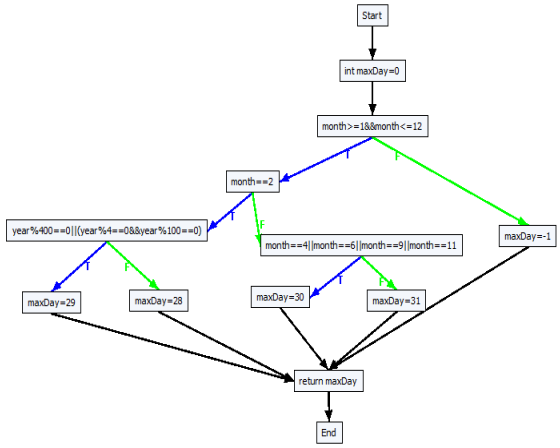

Figure 2. CFG of PUT getDayNum.

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 thesequence {[(*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 in 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`

PathID	Path's branch predications and their decisions
path1	[(*month* ≥ 1 &&*month* ≤ 12), T], [(*month* = 2), T], [(*year* % 400 = 0		(*year* % 4 = 0 &&*year* % 100 = 0)), T]
path2	[(*month* ≥ 1 &&*month* ≤ 12), T], [(*month* = 2), T], [(*year* % 400 = 0 || (*year* % 4 = 0 &&*year* % 100 = 0)), F]
path3	[(*month* ≥ 1 &&*month* ≤ 12), T], [(*month* = 2), F], [(*month*= 4|| *month*= 6|| *month*= 9 || *month*= 11), T]
path4	[(*month* ≥ 1 &&*month* ≤ 12), T], [(*month* =2), F], [(*month*= 4|| *month*= 6|| *month*= 9 || *month*=11), F]
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.

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 function for each branch predication and its decision of PUT getDayNum

Decision node	Fitness function	ID
[(*month* ≥ 1 &&*month* ≤ 12), T]	*fBchDist*(*month*, ≥, 1) + *fBchDist* (*month*, ≤, 12)	f_1T
[(*month* ≥ 1 &&*month* ≥ 12), F]	min(*fBchDist*(*month*, <, 1), *fBchDist*(*month*, >, 12))	f_1F
[(*month* = 2), T]	*fBchDist*(*month*, =, 2)	f_2T
[(*month* = 2), F]	*fBchDist*(*month*, ≠, 2)	f_2F
[(*year* % 400 = 0 || (*year* % 4 = 0 && *year* % 100 = 0)), T]	min(*fBchDist*(*year*%400, =, 0), (*fBchDist*(*year*%4, =, 0) + *fBchDist*(*year*%100, =, 0)))	f_3T
[(*year* %400 = 0 || (*year* % 4 = 0 && *year* % 100 = 0)), F]	*fBchDist*(*year* %400, ≠, 0) + min(*fBchDist*(*year* %4, ≠, 0), *fBchDist* (*year* %100, ≠, 0))	f_3F
[(*month*= 4 || *month*= 6 || *month*= 9 || *month*= 11), T]	min(*fBchDist*(*month*, =, 4), *fBchDist*(*month*, =, 6), *fBchDist*(*month*, =, 9), *fBchDist*(*month*, =, 11))	f_4T
[(*month*= 4 || *month*= 6 || *month*= 9 || *month*= 11), F]	*fBchDist*(*month*, ≠, 4) + *fBchDist*(*month*, ≠, 6) + *fBchDist*(*month*, ≠, 9) + *fBchDist*(*month*, ≠, 11)	f_4F

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 $a - b \leq 0$ 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 $b - a \geq 0$ then return 0 else return (abs($b - a$) + k)
14: **end switch**

Base onthese formulas, forcalculating fitness value for each branch predication, we generate the fitness function for each test path of the PUT getDayNum as below:

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

PathID	Test path fitness functions
path1	$F_1 = f_1T + f_2T + f_3T$
path2	$F_2 = f1T + f2T + f3F$
path3	$F_3 = f1T + f2F + f4T$
path4	$F_4 = f1T + f2F + f4F$
path5	$F_5 = f1F$

4.3. Apply multithreading of 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 simultaneous multithreading of the PSO algorithm by defining PSO it as 1 class extends Thread class of Java as follows: public class PSOProcess extends Thread

The multithreading of PSO can be executed through below algorithm:

Algorithm 5: Multithreading of Particle Swarm Optimization(MPSO)
Input: list of fitness functions Output:the set of test data that is solution to cover corresponding test path 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 PUTgetDayNum:

```
================================================
Test path ID:  2
Solution found at iteration 0, the solutions is:
     Best X1: 1404
     Best X2: 12
================================================
================================================
Test path ID:  4
Solution found at iteration 6, the solutions is:
     Best X1: 9069
     Best X2: 2
================================================
================================================
Test path ID:  1
Solution found at iteration 0, the solutions is:
     Best X1: 4750
     Best X2: 4820
================================================
================================================
Test path ID:  3
Solution found at iteration 247, the solutions is:
     Best X1: 33889
     Best X2: 6
================================================
================================================
Test path ID:  5
Solution found at iteration 386, the solutions is:
     Best X1: 8500
     Best X2: 2
================================================
```

Figure 4. Generated test data for the PUT getDayNum.

5. Experimental analysis

We compared 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 PUT of the given benchmark. Also we show our approach is better than state-of-the-art constraint-based test data generator Symbolic PathFinder [21].

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 (and also the fitness 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 side, 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:

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 advances the automatic ability in our proposal.

5.2. Path coverage ability

We also confirmed our proposed approach on the benchmark which is used in Mao's paper [9]. We performed in the environment of MS Windows 7 Ultimate with 32-bits and ran on

Intel Core i3 with 2.4 GHz and 4 GB memory. Our proposal was implemented in Java and run on the platform of JDK 1.8. We compared the coverage ability of all 8 programs in the benchmark as Table 5.

Table 5.The benchmark programs used for experimental analysis

PUT name	LOC	TPs	Args	Description
triangleType	31	5	3	Type classification for a triangle
calDay	72	11	3	Calculate the day of the week
cal	53	18	5	Compute the days between two dates
remainder	49	18	2	Calculate the remainder of an integer division
computeTax	61	11	2	Compute the federal personal income tax
bessj	245	21	2	Bessel J_n function
printCalendar	187	33	2	Print the calendar of a month in some year
line	92	36	8	Check if two rectangles overlap

* **LOC**: Lines of code **TPs**: Test paths**Args**: Input arguments

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 MPSO 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:

$$SR = \frac{\sum(all\ test\ paths\ were\ covered)}{1000}$$

- 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 MPSO by 1000 times, and calculated the average coverage for each run. AC formula is calculated for each PUT as follows:

$$AC = \frac{\sum(coverage\ for\ each\ run)}{1000}$$

The detailed results of the comparison with PUT benchmark used by Mao [9] in 2 criteria are shown in the Table 6.

From Table 6 can be seen that there are 4 PUTs (triangleType,computeTax, printCalendar, line) 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 MPSO is run, each PSO can generate test data which can cover the test path it is assigned to. Also with the remaining 4 PUTs (calDay, cal, reminder, bessj), our experiments fully covered all test paths with the same results of Mao [9].

5.3. Compare to constraint-based test data generation approaches

In this section we point out our advancement of the constraint-based test data generation approaches when generating test data for the given program that contains native function calls. We compare to Symbolic PathFinder (SPF) [21], which is the state-of-the-art of constraint-based test data generation approaches. Consider asample Java program as below:

```
int foo(double x, double y) {
int ret = 0;
if ((x + y + Math.sin(x + y))
== 10) {
ret = 1; // branch 1
}
return ret;
}
```

Due to the limitation of the constraint solver used in SPF, it cannot solve the condition((x + y + Math.sin(x + y)) == 10).Because this condition contains the native function Math.sin(x + y) of the Java language, SPFis unable to generate test data which can cover branch 1.

In contrast, by using search-based test data generation approach, for the condition((x + y + Math.sin(x + y)) == 10), we appliedKorel's formulain Table 1 to create fitness functionf_1T = abs((x + y + Math.sin(x + y)) - 10). Then using PSO to generate test data that satisfies this condition, we got the following result:

```
================================================
Solution found at iteration 486, the solutions is:
Optimation value = 0.000000
     Best X1: 477.906134
     Best X2: -466.906144
================================================
```

Figure 5. Generated test data for the condition which contains native function.

Table 6. Comparison between Mao's approach and MPSO

Program under test	Success rate (%)		Average coverage (%)	
	Mao[9]'s PSO	MPSO	Mao[9]'s PSO	MPSO
triangleType	99.80	100.0	99.94	100.0
calDay	100.0	100.0	100.0	100.0
cal	100.0	100.0	100.0	100.0
remainder	100.0	100.0	100.0	100.0
computeTax	99.80	100.0	99.98	100.0
bessj	100.0	100.0	100.0	100.0
printCalendar	99.10	100.0	99.72	100.0
line	99.20	100.0	99.86	100.0

6. Conclusion

This paper has introduced and evaluated a combination static program analysis and PSO approach for evolutionary structural testing. We proposed a method which uses a fitness function for each test path of a PUT, and then executed 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 [9] test data generation method using PSO in terms of both automatic and coverage ability for a PUT. Our approach also addressed a limitation of constraint-based test data generation approaches, which generate test data for conditions that contain native functions.

As future works, we will continue to extend our proposal to be applicable to many kinds of UTs, such as PUTs which contain calls to other native functions or PUTs that handle string operations or complex data structures. In addition, further research is needed to be able to apply this proposal for programs not only inacademics but also in industry.

References

[1] B. Antonia, "Software Testing Research: Achievements, Challenges, Dreams", 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] B. W. Kernighan and P. J. Plauger, "The Elements of Programming Style", McGraw-Hill, Inc, New York (1982).

[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] A.Windisch and S.Wappler, "Applying particle swarm optimization to software testing", 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", 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] J.Kennedy, and R.Eberhart, "Particle swam optimization", Proceedings of IEEE International Conference on Neural Networks (ICNN'95), pp. 1942–1948 (1995)

[12] Robert Gold, "Control flow graph and code coverage", Int. J. Appl. Math. Comput. Sci., Vol. 20, No. 4, 2010, pp. 739-749

[13] Bryan F. Jones, Harmen-Hinrich Sthamer, and D.E. Eyres, "Automatic structural testing using genetic algorithms", Software Engineering, 11(5):299–306, September 1996.

[14] M.Harman, P.McMinn, "A theoretical and empirical study of search-based testing: local, global, and hybrid search", IEEE Trans. Softw. Eng. 36(2), 226–247 (2010)

[15] Agrawal K., Srivastava G, "Towards software test data generation using discrete quantum particle swarm optimization", ISEC, Mysore, India (February 2010).

[16] S. Tiwari, K.K. Mishra, A.K. Misra, "Test case generation for modified code using a variant of particle swarm optimization (PSO) Algorithm [C]", Proceedings of the Tenth IEEE International Conference on Information Technology: New Generations (ITNG), 2013, pp. 363–368.

[17] X.M. Zhu, X.F. Yang, "Software test data generation automatically based on improved adaptive particle swarm optimizer", Proceedings of the International Conference on Computational and Information Sciences, 2010, pp. 1300–1303.

[18] S. Dahiya, J. Chhabra, S. Kumar., "PSO based pseudo dynamic method for automated test case generation using interpreter", Proceedings of the Second International Conference on Advances in Swarm intelligence, 2011, pp. 147–156.

[19] S. Singla, D. Kumar, H.M. Rai, P. Singla, "A hybrid PSO approach to automate test data generation for data flow coverage with dominance concepts", Int. J. Adv. Sci. Technol. 37 (2011).

[20] E. J. Weyuker, "The applicability of program schema results to programs", International Journal of Parallel Programming, vol. 8, 387--403 (1979).

[21] C. S. Pasareanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, N. Rungta, "Symbolic PathFinder: Integrating Symbolic Execution with Model Checking for Java Bytecode Analysis", Automated Software Engineering Journal, Springer (2013).

[22] G. M. Michael, M. Schatz, "Generating software test data by evolution", IEEE Transactions on Software Engineering, vol. 27, 1085--1110 (2001).

[23] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing", Information and Software Technology, vol. 43, 841--854 (2001).

[24] J. Wegener, B. Kerstin, and P. Hartmut, "Automatic Test Data Generation For Structural Testing Of Embedded Software Systems By Evolutionary Testing", Genetic and Evolutionary Computation Conference. Morgan Kaufmann Publishers Inc. (2002).

[25] S. Levin and A. Yehudai, " Evolutionary Testing: A Case Study", Hardware and Software, Verification and Testing, 155--165 (2007).

[26] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios, "Application of genetic algorithms to software testing (Application des algorithmes genetiques au test des logiciels)", 5th International Conference on Software Engineering and its Applications, pp. 625--636. Toulouse, France (1992).

[27] W. Joachim, Andr, Baresel, and S. Harmen, "Suitability of Evolutionary Algorithms for Evolutionary Testing", 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment. IEEE Computer Society, Washington (2002).

[28] Duc-Anh Nguyen, Pham Ngoc Hung, Viet-Ha Nguyen, "A method for automated unit testing of C programs", Proceedings of 2016 3rd National Foundation for Science and Technology Development Conference on Information and Computer Science (NICS), 2016, pp. 17-22k.