XXXII NATIONAL SYSTEMS CONFERENCE, NSC 2008, December 17-19, 2008

TOWARDS SOFTWARE TEST DATA GENERATION USING BINARY PARTICLE SWARM OPTIMIZATION

Khushboo Agarwal, Ankur Pachauri, and Gursaran[1]

Abstract—**The use of metaheuristic search techniques for software test data generation has been explored by many researchers. In this paper we describe an investigation into the application of Binary Particle Swarm Optimization (BPSO) for test data generation and compared its performance with a simple Genetic Algorithm (GA). Preliminary results of an experiment show that the performance of a BPSO may be better as compared to a GA. Performance data for BPSO was also collected for different values of *Vmax*, a control parameter for BPSO. It was observed that for small population sizes the performance improved as the value of *Vmax* was decreased.**

Index Terms— **Software Testing, Automated Software Test Data Generation, Binary Particle Swarm Optimization, Genetic Algorithms.**

I. INTRODUCTION

SOFTWARE testing is a critical step in the software development process. It is defined as the process of executing a software system to determine compliance with specifications [1]. During testing, the software to be tested is executed on a *test set* of *test cases* (or *test data*) - a specific point in the input domain - and the results are evaluated.

An adequacy criterion [2] specifies *test requirements* – specific items from specification or code that must be covered during testing. It, however, does not specify how test sets are to be generated to meet the specified requirements. Test sets can be generated manually, but this may become difficult. So there is a need for automatic test data generation. Although the problem of test data generation is known to be undecidable, one paradigm that has been actively researched is the *dynamic test data generation* paradigm [3]. In dynamic test data generation, the problem of test data generation is reduced to that of function minimization or maximization. The source code is instrumented to collect information about the program as it executes. Collected information is used to heuristically measure how close the test data is to satisfying the test requirements. The measure is then used to modify the input parameters to progressively move towards satisfying the test requirement. Recently the use of metaheuristic techniques has been explored for this purpose [4, 5].

In this paper we have explored the application of Binary Particle Swarm Optimization (BPSO) for software test data generation and compared its performance with a simple Genetic Algorithm (GA).

This paper is organized as follows: Section II presents related research in search based software test data generation. Section III briefly describes the search techniques of GAs and PSO along with the binary PSO. Section IV outlines a general approach to software test data generation with metaheuristic search techniques that has been used in this research. Section V describes the experiment carried out in this research and presents the results. Section VI summarizes the paper and gives suggestions for the future work.

II. RELATED RESEARCH

According to McMinn [4], "Metaheuristic search techniques are high-level frameworks which utilize heuristics in order to find solutions to combinatorial problems at a reasonable computational cost. Such a problem may have been classified as NP-complete or NP-hard, or be a problem for which a polynomial time algorithm is known to exist but is not practical." Search techniques such as Simulated Annealing, Evolutionary Algorithms [4] and other new ideas in optimization such as Particle Swarm Optimization (PSO) [6] are metaheuristic in nature. To the problem of test data generation, metaheuristic techniques such as Simulated Annealing, Evolutionary Algorithms have been widely used [4, 5, 11]. Recently, techniques such as particle swarm optimization [7] have also been used.

Windisch et al. [7] have explored the application of a Comprehensive Learning Particle Swarm Optimizer (CL-PSO) for software test data generation and compared its performance with Genetic Algorithms. Their results show that PSO outperforms GAs for most code elements to be covered in terms of effectiveness and efficiency. In the research described in this paper we have explored the application of Binary PSO to test data generation. An experiment has been carried out on a program that required a novel test case design and comparisons have been made with a Genetic Algorithm. We have also explored the impact of an important PSO control parameter on test data generation.

III. METAHEURISTIC SEARCH TECHNIQUES

This section presents a brief description of Genetic Algorithms and Binary Particle Swarm Optimization.

[1]The authors are with the Department of Mathematics, Faculty of Science, Dayalbagh Educational Institute, Dayalbagh, Agra 282 005.

A. Genetic Algorithms

Genetic algorithms (GAs) [8] are search algorithms that are based on the ideas of genetics and evolution in which new and fitter set of string individuals are created by combining portions of fittest string individuals of an older set. A genetic algorithm execution begins with a random *initial population* of candidate solutions $\{s_i\}$ to an objective function f(s). Each candidate s_i is generally a vector of parameters to the function f(s) and usually appears as an encoded binary string (or *bit string*) called a *chromosome*. An encoded parameter is referred to as a *gene*, where the parameter's values are the gene's *alleles*. The number of parameters and the size of the bit encoding of each parameter defines the *length* of the chromosome. In this paper a chromosome represents an encoding of a test case.

After creating the initial population, each chromosome is evaluated and assigned a *fitness* value. Evaluation is based on a *fitness function* that is problem dependent. From this initial selection, the population of chromosomes iteratively evolves to one in which candidates satisfy some termination criteria or, as is also in our case, fail to make any progress. Each iteration step is also called a *generation*.

Each generation may be viewed as a two stage process. Beginning with the *current population*, *selection* is applied to create an *intermediate population* and then *recombination* and *mutation* are applied to create the *next population*. The most common selection scheme is the *roulette-wheel selection* in which individual chromosomes are chosen using "stochastic sampling with replacement" to construct the intermediate population. Additionally with *elitism* the fittest chromosomes survive from one population to the other.

After selection, *crossover*, i.e., recombination, is applied to randomly paired strings with a probability. Amongst the various crossover schemes are the *one point*, *two point* and the *uniform crossover* schemes. In addition to crossover, *mutation* is used to prevent permanent loss of any particular bit or allele. Mutation application also introduces genetic diversity. Mutation results in the flipping of bits in a chromosome according to a mutation probability which is generally kept very low.

The chromosome length, population size, and the various probability values in a GA application are referred to as the *GA parameters* in this paper. Selection, crossover, mutation are also referred to as the *GA operators*.

B. Particle Swarm Optimization

Particle Swarm Optimization (PSO) was initially proposed to find optimal solutions of the continuous space problems by Kennedy and Eberhart [6, 9] in 1995. In PSO the search starts with a randomly generated population of solutions called the swarm of particles in *d*-dimensional solution space. Particle *i* is represented as $X_i = (x_{i1},x_{i2},.....,x_{id})$ which is called the position of the particle *i* in d-dimensional space. With every particle *i* a velocity vector $V_i = (v_{i1},v_{i2},.....,v_{id})$ is associated that plays an important role in deciding next position of particle and is updated in each iteration. For updating the velocity of each particle, the particle's best $P_{ibest}=(p_{i1},p_{i2},.....,p_{id})$ which is the best position of particle *i* achieved so far and global best $P_{gbest}=(p_{g1},p_{g2},....p_{gd})$ which is the best position of the swarm achieved so far by any particle of the swarm, are used. Following equations (1) and (2) are used to find new velocity and position of particle *i* in iteration *t+1*.

$$V_i(t+1)=w.V_i(t)+c_1\varphi_1(p_{ibest}-X_i(t))+c_2\varphi_2(p_{gbest}-X_i(t)) \quad (1)$$

$$X_i(t+1)=X_i(t)+V_i(t+1) \quad (2)$$

In equation (1), *w* is the inertia weight which controls the impact of previous history of velocity on global and local search abilities of particles [9], c_1 and c_2 are positive learning constants which determine the rate by which the particle moves towards individual's best position and the global best position. Usually, c_1 and c_2 are chosen in a way so that there sum doesn't exceed 4. If it exceeds 4 at any time then both the velocities and positions will explode toward infinity. φ_1 and φ_2 are random numbers drawn from uniform probability distribution of *(0,1)*. In this way positions and velocities of the particles are evolved in each iteration until the optimal solution is not obtained.

Binary Particle Swarm Optimization

In 1997 Kennedy and Eberhart [10] introduced the binary particle swarm optimization (BPSO) algorithm. In the binary version every particle is represented by a bit string and each bit is associated with a velocity, which is the probability of changing the bit to 1. Particles are updated bit by bit and velocity must be restricted within the range [0,1]. Let *P* be the probability of changing a bit from 0 to 1, then *1-P* will be the probability of not changing the bit to 1. This probability can be represented as the following function:

$$P(x_{id}(t)=1)=f(x_{id}(t),v_{id}(t-1),p_{id},p_{gd}) \quad (3)$$

where $P(x_{id}=1)$ is the probability that an individual particle *i* will choose *1* for the bit at the d^{th} site in the bit string, $x_{id}(t)$ is the current state of particle *i* at bit *d*, $v_{id}(t-1)$ is a measure of the string's current probability to choose a *1*, p_{id} is the best state found so far for bit *d* of individual *i*, i.e., a *1*or a *0*, p_{gd} is *1* or *0* depending on what the value of bit *d* in the global best particle.

The most commonly used measure for *f* is the sigmoid function which is defined as follows:

$$\mathrm{f}(\mathrm{v}_{\mathrm{id}}(t))=\frac{1}{1+e^{-v_{id}(t)}} \quad (4)$$

where,

$$v_{id}(t)=wv_{id}(t-1)+(\varphi_1)(p_{id}-x_{id}(t-1))+(\varphi_2)(p_{gd}-x_{id}(t-1)) \quad (5)$$

Equation (5) gives the update rule for the velocity of each bit, where φ_1 and φ_2 are random numbers drawn from the uniform distributions. Sometimes these parameters are chosen from the uniform distribution, such that their sum is 4. The *v* value is sometimes limited so that *f* does not approach *0.0* or *1.0* too closely. In this case, constant parameters *[Vmin, Vmax]* are used. When v_{id} is greater than *Vmax*, it is set to *Vmax* and if v_{id} is smaller to than *Vmin,* then V_{id} is set to *Vmin*. This simply limits the ultimate probability that bit x_{id} will take on a zero or one value. A higher value of *Vmax* makes new vectors less likely. Thus *Vmax* in the discrete particle swarm plays the role of limiting exploration after the population has converged [10], i.e., it can be said that *Vmax* controls the ultimate mutation rate or temperature of the bit vector. Smaller *Vmax* leads to a higher mutation rate [10]. This is explored in the experiment described in this paper.

C. PSO and GAs

According to Windisch [7], GAs are popular as they are able to solve efficiently non-linear, multimodal problems. They can be applied to solve both discrete and continuous space problems. In comparison, PSO is new optimization technique. It is also population based algorithm and can deal with continuous space multidimensional problems and with slight modifications, it can also be used for discrete space problems. A higher effectiveness and efficiency is generally attributed to PSO [7]. This is evaluated in this paper also.

IV. IMPLEMENTING SEARCH FOR TEST DATA GENERATION

The steps below outline a general format for the application of GA and BPSO for software test data generation assuming a binary string representation: Let P be the program for which test data is to be generated.

1. Choose an appropriate test adequacy criterion C.
2. Setup the Metaheuristic Search:
   a. Select a binary string representation for the test case to be input to program P.
   b. Select an evaluation (raw fitness computation) function.
   c. Instrument P using C to create program P_t. Instrumentation is necessary to measure coverage. Program P_t is then used directly for test data generation.
   d. Select suitable search parameters.
3. Generate test data.
   a. Run the metaheuristic search for test data generation using P_t for fitness computation.
   b. Identify and eliminate infeasibility.
   c. Regenerate test data if necessary.

In dynamic test data generation, test data is generated to meet the requirements of a particular test adequacy criterion. The entire setup depends on this choice. In this paper, C is taken to be the branch coverage criterion [2]. A search setup begins with the design of a string representation for a test case and the identification of a suitable fitness function. With branch coverage, raw fitness is evaluated based on the predicates in branching conditions [4,5, 11]. Assuming that fitness is being *maximized*, for each branch, string fitness is evaluated as: Fitness = $1/(|h-g|+\delta)$, where δ is a small quantity (approximately 0.0001) to prevent numeric overflow, h and g represent the desired and the actual branch predicate value respectively. Here, if the branching condition is not reached, the fitness is assumed to be very low. Program P is suitably instrumented to give P_t so as to mark the coverage of branches as they are traversed in an execution of P_t and to return a computed fitness value for the test case.

Infeasibility may prevent test data from being generated to satisfy C. It may be dealt with as follows. If the search is attempting to traverse a particular branch, but is unable to do so over a sufficiently large, predetermined, number of iterations, then the search run is aborted and the branch is manually examined for infeasibility. How this may be done is not discussed here. If the branch is found to be infeasible then it is marked as traversed and the search is rerun. After a traversal of all the branches has been attempted, the test data generation process outputs the test set together with the percentage of test branches that are satisfied by the test set.

V. EXPERIMENT

An experiment in test data generation with BPSO and GA was carried out on an implementation of a Soda Vending Machine simulator, adapted from [11], with procedural flow of control. This is described in this section.

A. The Soda-Vending Machine

The flow chart in Fig. 1 describes the operation “buy Soda” in a Soda Vending Machine. The flow chart includes one primary and two alternate scenarios. In the primary scenario, the customer makes a cash (money) input (taken to be an integer amount here), and then makes a soda selection (an integer value). The soda machine then dispenses a cold can of the selected soda. The alternate scenarios are the “out of soda selection” scenario and the “out of change” scenario. In an occurrence of any of the alternate scenarios, the current transaction is terminated and the money is returned. “Out of soda” scenario occurs when the soda type, the customer demands, is not in the menu or not present in the inventory. The other scenario “out of change scenario” occurs when the required change is not available in the cash reserved for returning the change. The Soda Vending Machine parameters are listed in Table I.

Since there can be a variable number of transactions after the inventory in the vending machine is restocked and that all the scenarios need to be covered, the test case must be designed to reflect this requirement. Each test case is a sequence of *variable* number of (soda type, money) inputs to the machine so that all the scenarios can be executed. Fig. 2 illustrates the binary string representation of a test case [11]. Although the binary string is of a fixed length, the actual portion of the string used, i.e., the count of (soda type, money) inputs, is decided by the value encoded in the initial part of the string as shown in Fig. 2.

B. *Experiment*

A number of experiments on the test data generation for the example of Soda Vending Machine software, implemented in C, were carried out with both Genetic Algorithms and Binary PSO. The chosen test adequacy criteria was branch coverage and the computation of fitness was instrumented as described in the previous section. The different GA and BPSO parameters used in the experiments are described in Table II. A hundred experiments were carried out for each of different population sizes in the range 5 – 50 and the average number of iterations required to achieve 100% coverage was computed.

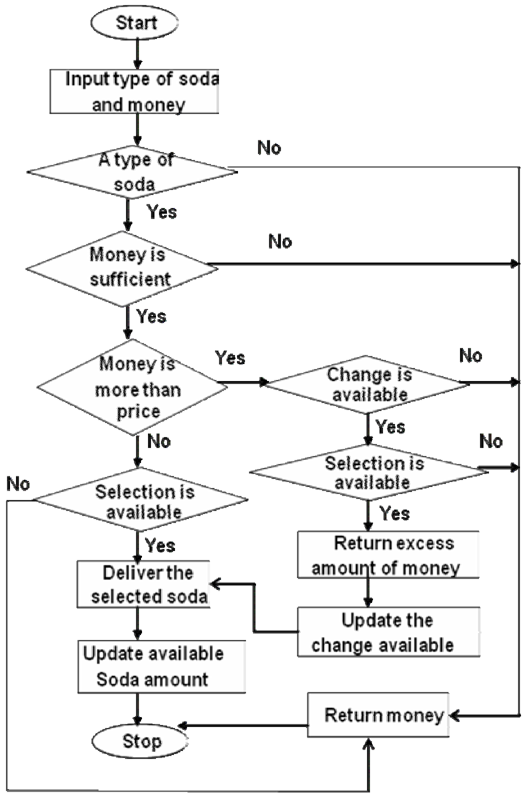


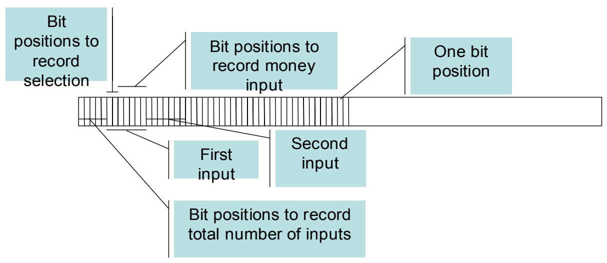
Fig. 1. Flow chart of the operation of a Soda Vending Machine.

TABLE I
SODA VENDING MACHINE PARAMETERS

Number of Types of Soda	3, Labeled 0, 1 and 2
Cost of each Soda Type	0 – Rs. 5, 1 – Rs. 10 and 2 – Rs. 15
Initial quantity of each Soda Type	7
Denominations	5 Labeled 0 through 4, 0 – Rs. 20, 1 – Rs.10, 2 – Rs. 5, 3 – Rs.2, 4 – Rs.1
Initial Quantity of Notes of Denominations 0 – 4	6

TABLE II
DIFFERENT BPSO AND GA PARAMETERS.

	Binary PSO
Vmax, Vmin	7, 0
φ_1, φ_2	Random numbers from the uniform distribution (0,4), such that $\varphi_1+\varphi_2 \leq 4$
w	[0.5+ (rnd/2.0)], where rnd is random number drawn from uniform distribution (0,1).
	Genetic Algorithm
Crossover	Two Point; Probability – 0.8
Mutation	Probability – 0.001



Bit positions to record selection
Bit positions to record money input
One bit position
First input
Second input
Bit positions to record total number of inputs

Fig. 2 Binary string representation of a test case

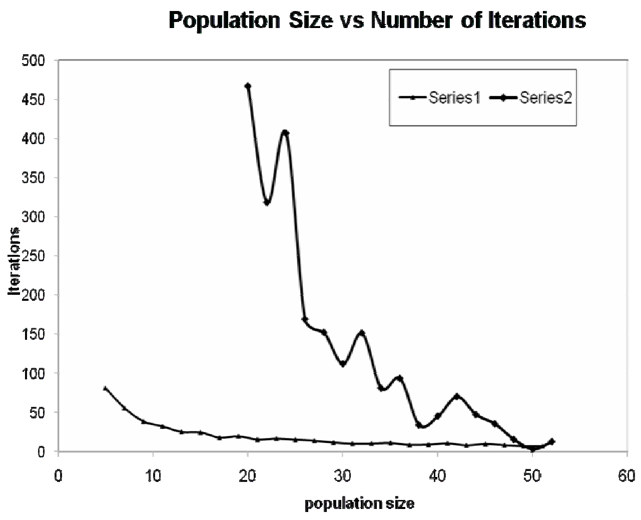


Fig. 3. plot between different population sizes and number of iterations to achieve full branch coverage with BPSO (Series 1) and GA (Series 2)

C. *Results*

There are total 27 branches in the program code of the Soda Vending Machine simulator. The plots of the results of experiments are shown in Fig. 3. We note that as the population size increases both GA and BPSO applications require fewer iterations to achieve coverage. BPSO requires very few iterations even with a very small population. The GA performance, however, degrades considerably with smaller population sizes. In fact, with a population size of 5, the GA required up to 4000 iterations to achieve coverage. This may have occurred since with a smaller population size exploration capability of a GA may have been reduced.

In Section III it was mentioned that *Vmax* controls the ultimate mutation rate or temperature of the bit vector and that smaller *Vmax* leads to a higher mutation rate [10]. With higher mutation rates the exploratory capability of the system increases and this may have an impact on test data generation. This was explored with different values of *Vmax*. Fig. 4 shows the outcome of the experiments. Clearly, with a smaller population size, decreasing the value of *Vmax* improves the performance of the BPSO. In other words as the exploratory capability of the system increases, fewer iterations are required to achieve coverage. This, however, needs to be further verified with more experiments.

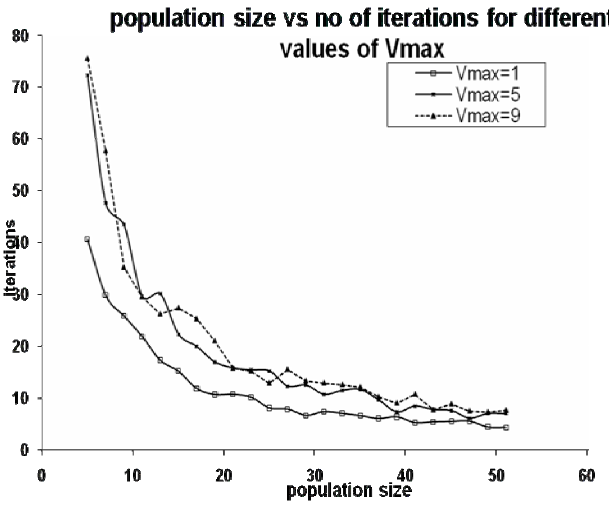


Fig. 4 Plot for different *Vmax* values

VI. Conclusion and Future Work

In this paper we have shown the application of a Binary PSO for software test data generation and compared the test data generation capability with a Genetic Algorithm on a simple Soda Vending Machine simulator. It was seen that the performance of the BPSO was much better than the GA with small population sizes. It was also observed that as *Vmax* was decreased, fewer iterations were required to achieve coverage.

There are many variants of the canonical PSO technique that have been described in the literature, the application and comparison of these need to be carried out. Moreover, given the number of control parameters of a BPSO, experiments need to be carried out over different parameter settings. Furthermore, the performance of BPSO can also be compared with other metaheuristic techniques.

References

[1] G.M. Kapfhammer, “Software testing,” *The Computer Science and Engineering Handbook*, CRC Press. May 2004.

[2] H. Zhu., A. V. Patrick, Hall, and H. R. John, “Software unit test coverage and adequacy,” *ACM Computing Surveys*, vol. 29, no. 4, pp. 366–427, 1997.

[3] B. Korel, “automated software test data generation,” *IEEE Trans. Software Eng.,* vol. 16, no. 8, pp. 870-879, 1990.

[4] P. McMinn, “Search-based software test data generation: a survey,” *Software Testing, Verification and Reliability*, vol. 14, no.2, pp.105-156, 2004.

[5] P. McMinn and M. Holcombe, “Evolutionary testing using an extended chaining approach,” *Evolutionary Computation*, vol.14, no.1, pp. 41-64, 2006.

[6] J. Kennedy and R. Eberhart, *“Particle swarm optimization,”* In Proc. of the IEEE Int. Conf. on Neural Networks, Piscataway, NJ, 1995, pp. 1942–1948.

[7] A. Windisch, S. Wappler and J. Wegener, “Applying particle swarm optimization to software testing,” In *Proceedings of the Genetic and Evolutionary Computation Conference* (GECCO 2007), London, UK, July 2007, pp. 1121- 1128.

[8] D.E. Goldberg, *Genetic algorithms*. New Delhi: Pearson Education, 1989.

[9] M. Dorigo and T. Stützle, “The particle swarm: social adaptation in information-processing system”, *New Ideas in Optimization*, McGraw-Hill, London, 1999, pp. 279-387.

[10] J. Kennedy and R.C. Eberhart. “A discrete binary version of the particle swarm algorithm”, IEEE International Conference on Systems, Man, and Cybernetics, 1997.

[11] Gursaran, R. Porwal, and R. Caprihan: ‘Towards test data generation for cluster testing using UML sequence diagrams and genetic algorithms’, *In proceeding of the Sixth International Conference on Information Technology*, CIT-2003, Bhubaneshwar, India, Dec 22-25, pp.61-66, 2003.