Evolutionary Neural Network Prediction for Cumulative Failure Modeling

M. Benaddy[1], M. Wakrim[1] & S. Aljahdali[2]
1 : Dept. of Math. & Info. Equipe MMS, Ibn Zohr University Morocco. benaddym@yahoo.fr
2: Taif University Saudi Arabia

Abstract: **An evolutionary neural network modeling approach for software cumulative failure prediction based on feed-forward neural network is proposed. A real coded genetic algorithm is used to optimize the mean square of the error produced by training a neural network established by Aljahdali S. [3]. In this paper we present a real coded genetic algorithm that uses the appropriate operators for this encoding type to train feed-forward neural network. We describe the genetic algorithm and we also experimentally compare our approach with the back propagation learning algorithm for the regression model order 4. Numerical results show that both the goodness-of-fit and the next-step-predictability of our proposed approach have greater accuracy in predicting software cumulative failure compared to other approaches.**

Keywords: *Genetic Algorithms, Real Coded Genetic Algorithms, Feed-forward Neural Networks, Software Reliability.*

I. Introduction

Software reliability is defined as the probability of failure free software operation for a specified period of time in a specified environment [1]. Society's reliance on large complex systems mandates high reliability. Reliable software is a necessary component. Controlling faults in software requires that one can predict problems early enough to take preventive action. In the past 35 years more than 100 software reliability models have been developed to solve reliability models [19]. Most of these models as the models of software reliability growth depend on a certain a priori assumptions about the nature of software faults and the stochastic behavior of software process [5]-[6]. As a result, different models have different predictive performance at different testing phases across various projects. A single universal model that can provide highly accurate predictions under all circumstances without any assumptions is most desirable [22]-[14]. Neural network approach has proven to be a universal approximator for any non-linear continuous function with an arbitrary accuracy [6]-[16]-[17]-[18]. Consequently, it has become en alternative method in software reliability modeling, evolution and prediction. Karunanithi et al. [14]-[15] were the first to propose using neural network approach in software reliability prediction. Aljahdali et al. [4]-[3], Adnan et al. [2], Park et al. [22] and Liang et al. [17]-[18] have also mad contributions to software reliability predictions using neural networks, and have gained better results compared to the traditional analytical models with respect to predictive performance.

The most popular training algorithm for feed-forward neural networks is the back-propagation algorithm, the back propagation learning algorithm provides a way to train multilayered feed-forward neural networks [3] but the optimal training of neural network using conventional gradient-descent methods is complicated due to many attractors in the state space. In this paper we have developed a real coded genetic algorithm (RCGA) as an alternative to train the neural network that optimizes the error made by the neural network.

II. Software Reliability Data set

The Software Reliability Dataset was compiled by John Musa of Bell Telephone Laboratories [7]. His objective was to collect failure interval data to assist software managers in monitoring test status and predicting schedules and to assist software researchers in validating software reliability models. These models are applied in the discipline of Software Reliability Engineering. The dataset consists of software failure data on 16 projects. Careful controls were employed during data collection to ensure that the data would be of high quality. The data was collected throughout the mid 1970s. It represents projects from a variety of applications including real time command and control, word processing, commercial, and military applications. In our case, we used data from three different projects. They are Military, Real Time Control and Operating System. The failure data were initially stored in arrays, ordered by day of occurrence so that it could be processed.

III. Neural Network Architecture

The architecture of the network used for modeling software reliability problem is a multi-layer feed-forward network. It consists of an input layer, one hidden layer and an output layer [4]-[3]. The input layer contains a number of neurons equal to the number of delayed measurements allowed to build neural networks model in our case, there are four inputs to the network, they are *C(k-1), C(k-2), C(k-3), and C (k-4). C (k-1)* is the observed faults one day before the current day. The hidden layer consists of two nonlinear neurons and two linear neurons. The output layer consists of one output neuron producing the estimated value of the fault. There is no direct connection between the network input and output. Connections occur only through the hidden layer. The hidden units are fully connected to both the input and output. The structure of the adopted neural network is shown in figure 1.

978-1-4244-3806-8/09/$25.00 © 2009 IEEE

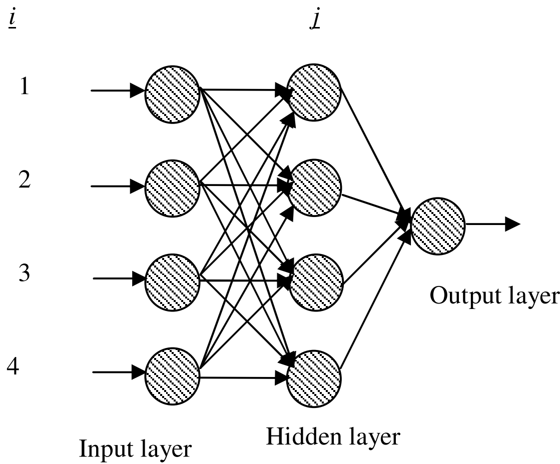

Figure 1: Feed-forward neural network structure

IV. THE GENETIC ALGORITHM

The genetic algorithm was developed and formalized by Holland [13]. It was further developed and shown to have wide applicability by Goldberg [10]. Schaffer, et al. [24] showed that it could be used to improve the learning ability of neural networks for simple pattern discrimination on a small data set. Evolving the weight set for a neural net with the inverted error as a fitness function has also been studied [12].Ways of combining GAs with NNs to form improved hybrid algorithms constitute a major research direction. For a good introduction and examination of GAs, see Michalewicz [20].

Although there are many possible varieties on the basic GAs, the operational of every genetic algorithm is described in the following steps:

1. Randomly create an initial population of chromosomes.
2. Compute the fitness of every member of the current population.
3. If there is a member of the current population that satisfies the problem requirements then stop. Otherwise continue to the next step.
4. Create an intermediate population by extracting members from the current population using a selection operator.
5. Generate a new population by applying the genetic operators of crossover and mutation to this intermediate population.
6. Go back to step 2.

V. REAL CODED GENETIC ALGORITHMS

The most common representation in GAs is binary [11]. The chromosomes consists of a set of genes, which are generally characters belonging to an alphabet {0, 1}. Therefore, a chromosome is a vector x consisting of l genes c_i: $x=(c_1,c_2,\ldots,c_l)$, $c_i=\{0,1\}$, Where l is the length of the chromosome.

However in the optimization problems of parameters with variables in continuous domains, it is more natural to represent the genes directly as a real numbers since the representation of solution are very close to the natural formulation, i.e. there are no differences between the genotype and the phenotype. The use of this real-coding in numerical optimization on continuous domains appears in Michalewicz [20].

In this case, a chromosome is a vector of floating point numbers. The chromosome length is the vector length of the solution of the problem; thus, each gene represents a variable of the problem. The gene values are forced to remain in the interval established by the variables they represent, so many genetic operators are developed for them, such as, Flat crossover [23], Arithmetic crossover [21] and BLX-α crossover [9] for the crossover operators and Random mutation and non-uniform mutation [21].

VI. THE REAL CODED GENETIC ALGORITHM TO TRAIN NEURAL NETWORK FOR SOFTWARE RELIABILITY PREDICTION

As mentioned above, real coding is the most suitable coding for continuous domains. Since our goal is feed-forward neural network training which predicts the cumulative future faults in the software, it appears logical to use this coding and genetic operators associated to it. Among the advantages of using real-valued coding over binary coding is increased precision. Binary coding of real-valued numbers can suffer loss of precision depending on the number of bits used to represent one number. Moreover, in real-valued coding chromosome string become much shorter. For real-valued optimization problems, real-valued coding is simply much easier and more efficient to implement, since it is conceptually closer to the problem space. In particular, our aim is to train a feed-forward NN to predict future faults in the software from the previous four discovered faults.

A chromosome consists of all the network weights. One gene of a chromosome represents a single weight value. In our case there are *4x4* weights for the input-layer plus *4x1* biases plus *4x1* weights plus *1x1* biases for the output-layer so, the length of the chromosome is *l= 4x4 + 4x1 + 4x1 + 1x1 = 25*. The weights and biases of the neural network are placed on a chromosome as shown in figure 2.

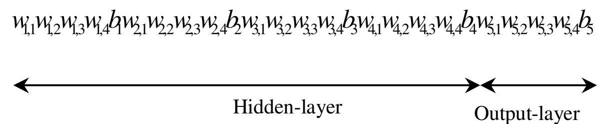

Figure 2: The chromosomal representation of the neural network

Fitness function: the fitness function should reflect the individual's performance in the current problem. We have chosen *1/(1+mse)* as a fitness function Eq. (2), where *mse* is the mean squared error during training defined in Eq (1).

$$mse = \frac{1}{n}\sum(\beta_i - \hat{\beta}_i)^2 \qquad \textbf{(1)}$$

Where n is the number of training faults used during the training process. β_i And $\hat{\beta}_i$ are the actual and the predicted output respectively during the learning process.

$$fitness = \frac{1}{1+mse} \qquad \textbf{(2)}$$

Selection mechanism: The roulette wheel selection is used to create the intermediate population. For each chromosome C_i in a population P, the probability $p_s(C_i)$, of including a copy of this chromosome in the intermediate population P' is calculated as in Eq. (3)

$$p_s(C_i) = \frac{fitness(C_i)}{\sum_{j=1}^{P} fitness(C_j)} \quad (3)$$

Where P is the number of individuals in the population P.

Creating a new generation by applying the genetic operators to the intermediate population. Once the intermediate population is created, the next step is for the population of the next generation by applying the crossover and mutation operators on the chromosomes in P'. Two chromosomes are randomly selected from this intermediate population and serve as parents. Depending upon a probabilistic chance p_c (crossover rate), it is decided whether these two will be crossed over. After applying these genetic operators, the resulting chromosome is inserted into the new population. This step is repeated until the new population reaches the population size less two individuals. Moreover, the two best individuals in the current population are included in the new population (elitist strategy) [8], to make sure that the best-performing chromosome always survives intact from one generation to the next. The crossover used is the BLX-α crossover with the crossover rate $p_c = 0.7$ and the parameter $\alpha = 0.5$. After the application of the crossover operator, each of the genes of the resulting chromosomes is subject to possible mutation, which depends on a probabilistic chance p_m, the mutation rate. The mutation operator used is non-uniform mutation with the mutation rate $p_m=0.06$ and the parameter $b=5$.

VII. THE REAL CODED GENETIC ALGORITHM TRAINING AND TESTING RESULTS

The initial weights were randomly chosen in the interval [0, 1]. For each project we performed a number of simulations with a population of 200 individuals and a maximum of generation equal to *Gmax*. After the training process the Normalize Root Square Error (*NRMSE*, see Eq. 4) is computed to compare the results obtained by real coded genetic algorithm with these obtained by the back-propagation learning algorithm.

$$NRMSE = \frac{1}{n}\sqrt{\frac{\sum_{i=1}^{n}(\beta(i)-\hat{\beta}(i))^2}{\sum_{i=1}^{n}(\beta(i))^2}} \quad (4)$$

The results of *NRMSE* obtained, by the back-propagation learning algorithm and the regression model in test phase is given in a table 1.

Table 1 : A comparison between Regression model order 4 and neural network model in testing case (NRMSE)[3].

Project Name	Military	Real Time Control	Operating System
Number of Faults	101	136	277
Training Data	71	96	194
Testing Data	101	136	277
Regression Model	3.1434	1.7086	1.0659
Neural Networks	1.0755	0.5644	0.7714

The results of *MSE* and *NRMSE* obtained, by the training with our real coded genetic algorithm in training and testing phases are given in table 2.

Table 2: Results for the *MSE* and NRMSE obtained using NNs trained by RCGA in the training and testing phases.

Project Name	Military	Real Time Control	Operating System
Number of Faults	101	136	277
Training Data	71	96	194
MSE	2.859155	2.0515463	2.0515463
NRMSE	2.0635e-4	5.3898e-4	3.4186e-5
Testing Data	101	136	277
MSE	4.2277226	2.6617646	3.0758123
NRMSE	4.7373e-5	3.1216e-4	1.5705e-5

In figure 3 to 14 we are showing the training, the error difference and the testing results for various projects using the neural network trained by our real coded genetic algorithm.

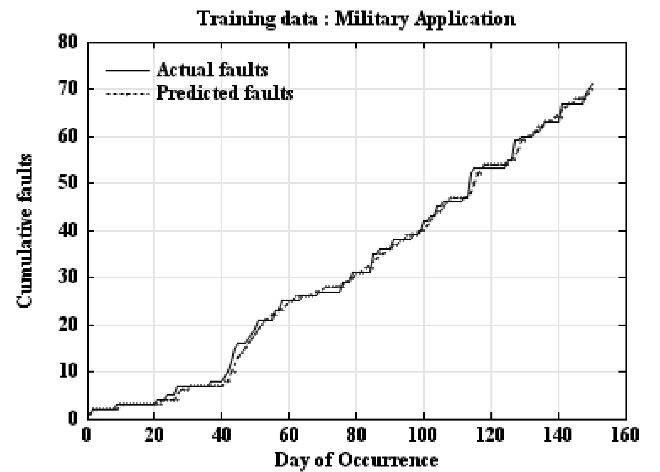

Figure 3: Actual and Predicted Faults in Training phase: Military Application.

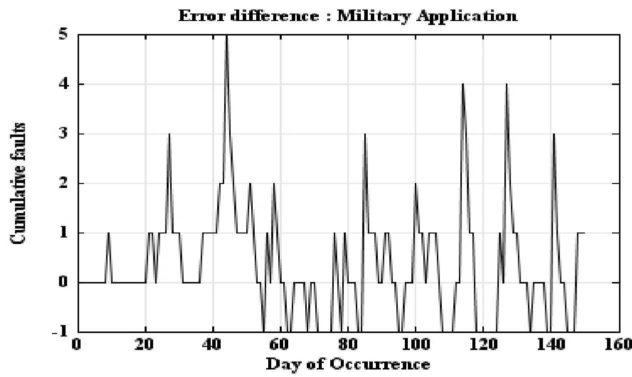

Figure 4: Prediction Error in training phase: Military Application.

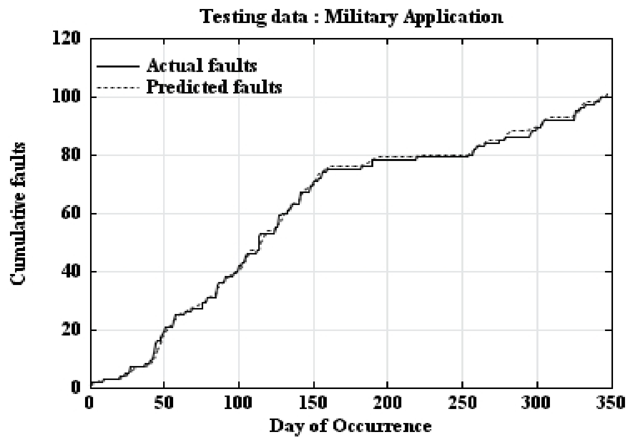

Figure 5: Actual and Predicted Faults in Testing phase: Military Application.

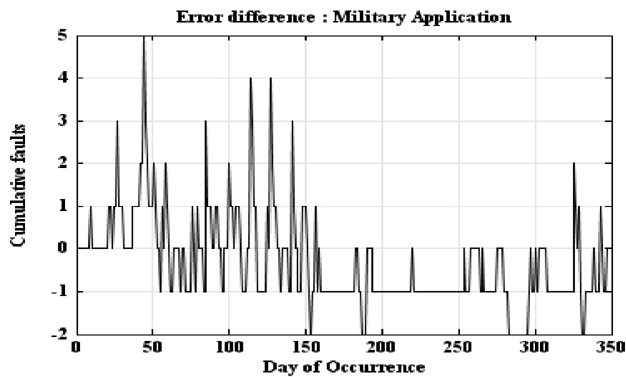

Figure 6: Prediction Error in testing phase: Military Application.

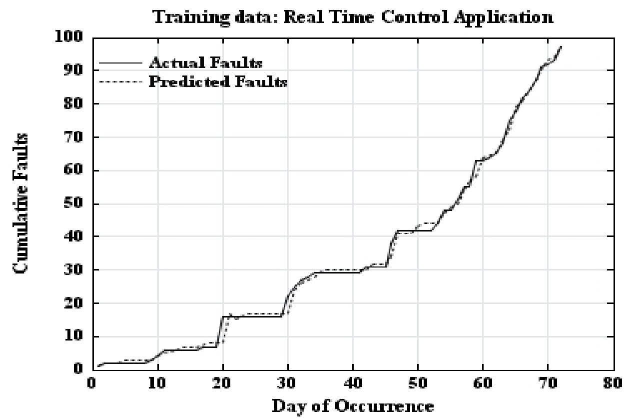

Figure 7: Actual and Predicted Faults in Training phase: Real Time Control.

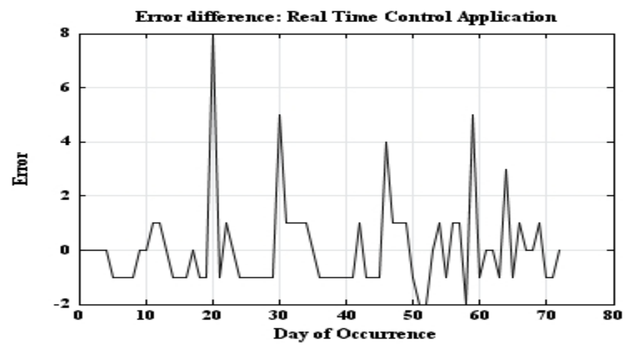

Figure 8: Prediction Error in training phase: Real Time Control.

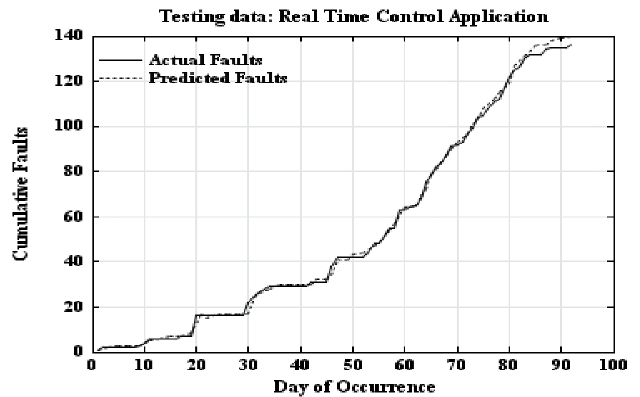

Figure 9: Actual and Predicted Faults in Testing phase: Real Time Control.

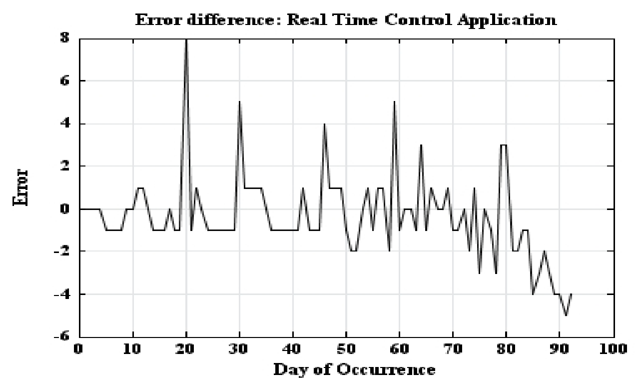

Figure 10: Prediction Error in testing phase: Real Time Control.

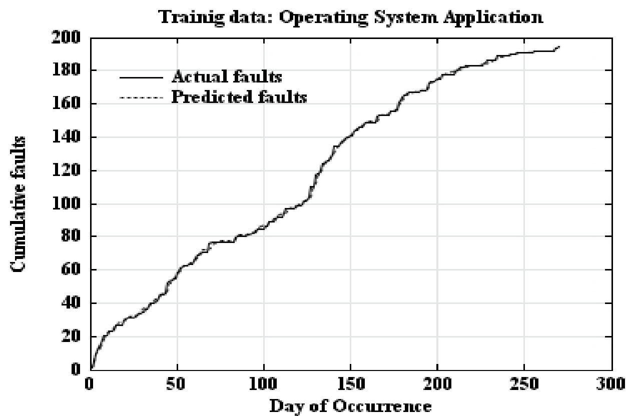

Figure 11: Actual and Predicted Faults in Training phase: Operating System

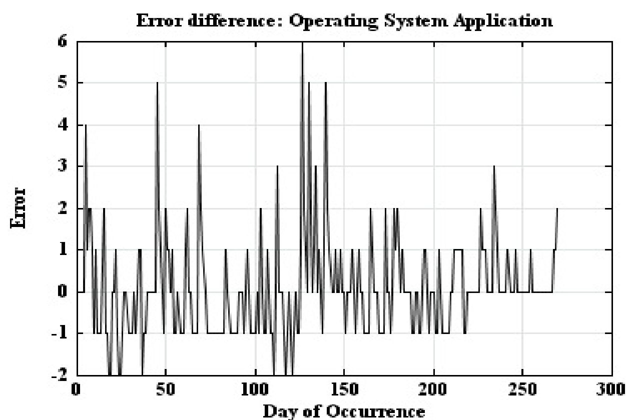

Figure 12: Prediction error in training phase: Operating System

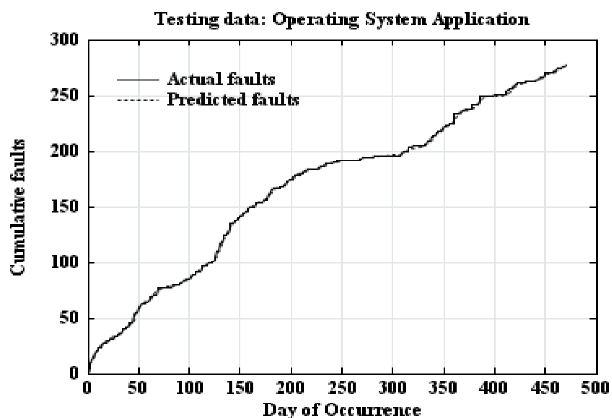

Figure 13: Actual and Predicted Faults in Testing phase: Operating System.

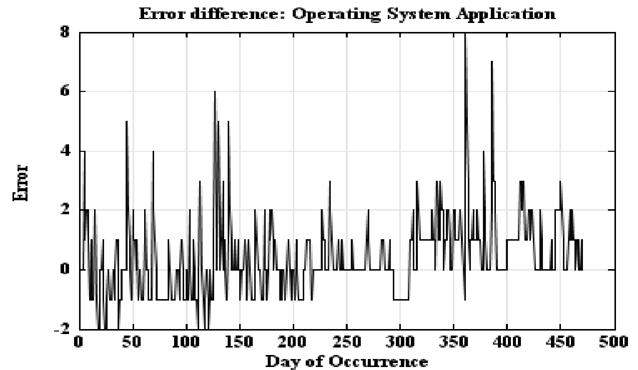

Figure 14: Prediction error in testing phase: Operating System.

VIII. CONCLUSION

In this paper, an evolutionary neural network modeling approach for software cumulative failure prediction is proposed. Genetic algorithm is used to learn the neural network by optimizing the mean square error produced by the neural network.

Experimental results show that our proposed approach adapts well across different projects, and has a better performance compared to the results obtained by neural network models for cumulative failure, learned by the back-propagation learning algorithm.

REFERENCES

[1] ANSI /IEEE, "Standard Glossary of Software Engineering Terminology," STD-729-1991, ANSI /IEEE, 1991.

[2] W.A. Adnan, M.H. Yaacob, “An integrated neural-fuzzy system of software reliability prediction.” In: Proceeding of the First International Conference on software Testing, Reliability and Quality Assurance, New Delhi, India, 1994.

[3] S. Aljahdali, K. A. Buragga, “Evolutionary Neural Network Prediction for Software Reliability Modeling” The 16th International Conference on Software Engineering and Data Engineering (SEDE-2007).

[4] S. Aljahdali, A. Sheta and D. Rine, “Prediction of Software Reliability: A Comparison between regression and neural network non-parametric Models”, Proceeding of the IEEE/ACS Conference, 25-29, June 2001.

[5] K.Y. Cai, C.Y. Xen, M.L. Zhang, “A critical review on software reliability modeling. Reliability Engineering Safety,” 1991.

[6] K.Y. Cai, L. Cai, W.D. Wang, Z.Y. Yu, D. Zhang, “On the Neural Network approach in software reliability modeling,” J Syst Software 2001.

[7] Data & Analysis Centre for Software DACS https://www.thedacs.com/databases/sled.

[8] K. De Jong “An Analysis of the Behavior of a class of Genetic Adaptive Systems.” Doctorate dissertation, Dept. of Computer and Communication Sciences, University of Michigan, Ann Arbor, 1975.

[9] L.J. Eshelman & J.D. Schaffer, “Real coded genetic algorithms and interval schemata” In L. Durrel Whitely, Foundation of genetic algorithms 2 (pp. 187-202). San Mateo: Morgan Kaufman.

[10] D.E Goldberg, “Genetic Algorithms in Search, Optimization and Machine Learning”. Addison Wisley New York, 1989.

[11] D.E. Goldberg, “Real-coded genetic algorithms. Virtual alphabets and blocking.” Complex Systems, 5, 1991, 139-167.

[12] R. Hochman et al., “Using the genetic algorithm to build optimal neural networks for fault-prone module detection” In Proc. the 7th Int. Symposium on Software Reliability Engineering 1996.

[13] J.H. Holand, “Adaptation in Natural and Artificial Systems”. Cambridge, Mass: MIT press, 1975.

[14] N. Karunanithi, D. Withtely, Y.K. Malaiya, "Prediction of Software Reliability using Connectionist Models," IEEE Trans Software Eng 1992.

[15] N. Karunanithi, D. Withtely, Y.K. Malaiya, "Using neural networks in reliability prediction," IEEE Software 1992.

[16] E.H.F. Leung, H.K. Lam, S.H. Ling, P.K.S. Tam, "Tuning of the structure and parameters of a neural network using an improved genetic algorithm," IEEE Trans Neural Networks 2003.

[17] T. Liang, N. Afzel, "Evolutionary neural network modeling for software cumulative failure time prediction," ELSEVIER Reliab Eng & Sys Safety 2005.

[18] T. Liang, N. Afzel, "On-line prediction of software reliability using an evolutionary connectionists model," ELSEVIER the Journal of Sys & Software 2005.

[19] M.R. Lyu, "Software Reliability Engineering: A Roadmap". Future of Software Engineering (FOSE'07) IEEE CS Press 2007.

[20] Z. Michalewicz, "Genetic Algorithms + Data Structures = Evolution Programs", Springer 1996.

[21] Z. Michalewicz, "Genetic Algorithms + Data Structures = Evolution Programs", New-York : Springer, 1992.

[22] J.Y. Park, S.U. Lee, J.H. Park, "Neural Network Modeling for Software Reliability Prediction from Failure Time Data," J Electr Eng Inform Sc 1999.

[23] N.J. Radcliffe, "Equivalence class of genetic algorithms" Complex Systems, 1991, 5 (2), 183-205.

[24] J. D. Schaffer, R. A. Caruana, and L.J. Eshelman, "Using genetic search to explicit the emergent behavior of neural networks." In S. Forrest (Ed.), Emergent Computation: Self-Organizing, Collective, and Cooperative Phenomena in Natural and Artificial Computing Networks (pp. 244-248). Cambridge, MA: MIT Press, 1991.