Assessing Software Reliability Growth with Different Combinations For Distributed Environment Using ANN Approach

P K Kapur1* Sunil K. Khatni2 Mashaallah Basirzadeh1
1Department of Operational Research, University of Delhi, India
2Mother Teresa Institute of Management, GGSIP University, Delhi

Abstract
Computer systems cover every aspect of our daily life. Although this has benefited the society but it has also made our lives more critically dependent on their correct functioning. Software reliability assessment is important to evaluate and predict the reliability and performance of software system. Several SRGM have been developed in the literature to estimate the fault content and fault removal rate per fault in software. Recently, artificial neural-network approach has been applied in estimating and predicting software reliability growth phenomenon. Computing has now reached the state of distributed computing. By amalgamating computers and networks into one single computing system and providing appropriate system software, a distributed computing system has created the possibility of sharing information and peripheral resources. Furthermore, these systems have improved performance of a computing system and individual users through parallel execution of programs, load balancing and sharing, and replication of programs and data. Distributed computing systems are also characterized by enhanced availability and increased reliability.

In this paper, we propose a neural-network based model in a distributed development environment. Proposed Model (with cases) assumes that the software system consists of a finite number of reused and newly developed sub-systems. The reused sub-systems do not consider the effect of complexity of the faults on the software reliability growth phenomenon because they stabilize over a period of time i.e. the growth is uniform whereas, the newly developed sub-system does consider that. Fault removal phenomena for reused and newly developed sub-systems have been modeled separately and are summed up to get the total fault removal phenomenon of the software system. Different cases of the proposed model have been validated, evaluated and compared by applying two actual data sets cited from real software development projects.

Keywords
Software Reliability Growth Model (SRGM), Artificial Neural Network (ANN), Non-Homogenous Poisson Process (NHPP), Distributed Computing, Fault Complexity

1 Introduction
Today, virtually every industry is highly dependent on computers for their basic functioning. With a continuous lowering cost and improved control, processors and software-controlled systems offer compact design, flexible handling and rich features. The size and complexity of computer-intensive systems has grown dramatically during the past decade and the trend will certainly continue in the future. The demand for complex hardware/software systems has increased more rapidly than the ability to design, implement, test and maintain them. When the requirements for and dependencies on computers increase, the possibility of crises from computer failures also increases. Software failures may be due to errors, ambiguities, oversights or misinterpretation of the specification that the software is supposed to satisfy, carelessness or incompetence in writing code, inadequate testing, incorrect or unexpected usage of the software or other unforeseen problems.

Software Reliability is an important attribute of software quality. While any system with a high degree of complexity, including software, will be hard to reach a certain level of reliability, system developers tend to push complexity into the software layer, with the rapid growth of system size and ease of doing so by upgrading the software. Software Reliability Engineering (SRE) is the quantitative study of the operational behavior of software-based systems with respect to user requirements concerning reliability.

Several software reliability growth models have been developed in the literature. Depending on the nature of growth phenomenon during testing, they are broadly classified in two categories. Some of these depict exponential reliability growth [3] where as others show S-Shaped growth [6, 21]. If the growth is uniform, generally exponential models have been used and for non-uniform growth S–Shaped models have been developed. Models that can capture variability in exponential and S-shaped curves have also been developed and they have been termed as flexible growth models [1, 5, 6, 15]. As S-shapedness in reliability can be ascribed to different reasons, many models exist in the literature at times leading to confusion in model selection from the plethora of models available.

Recent advances in neural networks show that they can be used in applications that involve estimation and prediction. Neural network methods may handle numerous factors and approximate any non-linear continuous function.

Proliferations of software reliability models have emerged as people try to understand the characteristics of how and why
software fails, and try to quantify software reliability. Over 200 models have been developed since the early 1970s, but how to quantify software reliability still remains largely unsolved. Many models are there and many more emerging, none of the models can capture a satisfying amount of the complexity of software; constraints and assumptions have to be made for the quantifying process. Therefore, there is no single model that can be used in all situations. No model is complete or even representative. One model may work well for a set of certain software, but may be completely off track for other kinds of problems. Many papers published in the literature address that neural networks offer promising approaches to software reliability estimation and prediction. Karunanithi et al. [9, 10, 11, 12] first applied neural network architecture to estimate the software reliability. They also illustrated the usefulness of connectionist models for software reliability growth predictions. Khoshgoftaar et al. [13] used the neural network as a tool for predicting the number of faults in a program and concluded that the neural networks produce models with better quality of fit and predictive quality. Cai et al [2] used the recent 50 inter failure times as the multiple-delayed inputs to predict the next failure time and found the effect of the number of neurons in the hidden layer and the number of hidden layers by independently varying the network architecture. Su et al. [19] have proposed a neural network based approach to software reliability assessment by combining various existing models in to a Dynamic Weighted Combinational Model (DWCM). Kapur et al. [8] develop a Generalized Dynamic Integrated Model (GDIM) for software having n different types of faults on basis of their severity. Zheng [22] have developed a non-parametric software reliability prediction system based on the neural network ensembles. Kapur et al. [7] have proposed a neural-network based logistic model for faults of different complexity.

The rest of this paper has been organized as follows: Section 2 briefly describes artificial-neural networks. In Section 3, artificial neural network methods are applied to build model for distributed development environment. Section 4, 5 and 6 provides the methods used for parameter estimation and comparison criteria. We used cumulative execution time as input and the corresponding cumulative faults as the desired output to form a training pair. Section 7 concludes the paper.

2 ARTIFICIAL NEURAL-NETWORKS

Neural networks are a computational metaphor inspired by studies of the brain and nervous system in biological organisms. They are highly idealized mathematical models of how we understand the essence of these simple nervous systems. A trained neural network can be thought of as an "expert" in the category of information it has been given to analyze. An artificial neural network involves a network of simple processing elements (artificial neurons), which can exhibit complex global behavior, determined by the connections between the processing elements and element parameters.

A typical feed-forward neural network comprises a layer of neurons called input layer that receive inputs (suitably encoded) from the outside world, a layer called output layer that sends outputs to the external world, and one or more layers called hidden layers that have no direct communication with the external world (figure 1). This hidden layer of neurons receives inputs from the previous layer and converts them to an activation value that can be passed on as input to the neurons in the next layer. The input layer neurons do not perform any computation; they merely copy the input values and associate them with weights, feeding the neurons in the first hidden layer. Feed-forward networks can propagate activations only in the forward direction. The input corresponds to the attributes measured for each training sample. The number of hidden layers is arbitrary. The weighted outputs of last hidden layer are input to units making up the output layer, which emits the network’s prediction for given samples [4].

![Figure 1 A multi-layer feed-forward neural network](image)

We can describe the neural network in a mathematical form. The neural network can be denoted as:

\[ o = \beta(i), \]  

(1)

where \( i = (i_1, i_2, i_3, \ldots, i_p) \) and \( o = (o_1, o_2, o_3, \ldots, o_q) \). The value of any \( o_k \) is given by

\[ o_k = \beta \left( b_k + \sum_{j=1}^{q} w_{jk} h_j \right), \quad k = 1, 2, \ldots, r \]  

(2)

where \( w_{jk} \) is the weight from hidden layer node \( j \) to output layer node \( k \), \( b_k \) is the bias of the node \( k \) in output layer, \( h_j \) is the output from node \( j \) of the hidden layer, and \( \beta \) is an activation function in output layer. The output value of the nodes in hidden layer is given by:

Copy Right © INDIACom-2009
\[ h_j = \alpha \left( b_j + \sum_{z=1}^{c} w_{zj}^j i_z \right), \quad j = 1, 2, \ldots, q \]  

where \( w_{zj}^j \) is the weight from input layer node \( z \) to hidden layer node \( j \), \( b_j \) is the bias of the node \( j \), \( i_z \) is the value in the input layer, and \( \alpha \) is an activation function in hidden layer.

The objective function \( o \) of the neural network can be considered as compounded function \( \beta(\alpha(i)) \).

Network design is a trial-and-error process and may affect the accuracy of the resulting trained network. There are no clear rules to the best number of hidden layer units the initial values of the weight may also affect the resulting accuracy. Once a network has been trained and its accuracy is not considered acceptable, it is common to repeat the training process with a different network topology or a different set of initial weights.

3 MODELING SOFTWARE RELIABILITY GROWTH

Neural network models have a significant advantage over analytical models, because they require only failure history as input and no assumptions. Consequently, they have drawn attention of many researchers in recent years. It has been found that neural network methods can be applied to estimate the number of faults and predict the number of software failures as they often offered better results than existing statistical analytical models. By deriving a compound function from the conventional statistical SRGM, we can build a neural network based SRGM.

ASSUMPTIONS

The neural-network based model for distributed development environment developed in this paper considers that software system consists of finite number of reused and newly developed components and takes into account the time lag between the failure and fault isolation / removal processes for the newly developed components.

The proposed models are based upon the following basic assumptions:

1. Failure observation / fault removal phenomenon is modeled by NHPP.
2. Software is subject to failures during execution caused by faults remaining in the software.
3. Software system consists of a finite number of reused and newly developed subsystems.
4. Software reliability growth in the reused sub-system is constant while in the newly developed sub-system is not.
5. Each time a failure is observed, an immediate effort takes place to decide the cause of the failure in order to remove it. The time delay between the failure observation and its subsequent removal is assumed to represent the complexity of fault. The more severe the fault more is the time delay.
6. During the fault isolation / removal, no new fault is introduced in the system.
7. Failure rate of the software is equally affected by faults remaining in the software.
8. The fault isolation / removal rate with respect to testing effort intensity is proportional to the number of observed failures whose cause is yet to be identified.

**NOTATIONS**

\( a \) Total fault content
\( a_i \) Initial fault content of type \( i \) reused component
\( a_j \) Initial fault content of type \( j \) newly developed components with hard faults
\( a_k \) Initial fault content of type \( k \) newly developed component with complex faults
\( b_i \) Proportionality constant failure rate / fault isolation rate per fault of \( i^{th} \) reused component
\( b_j \) Proportionality constant failure rate / fault isolation rate per fault of \( j^{th} \) newly developed component
\( b_k \) Proportionality constant failure rate / fault isolation rate per fault of \( k^{th} \) newly developed component
\( m_i(t) \) Mean number of faults removed from \( i^{th} \) reused component by time \( t \)
\( m_j(t) \) Mean number of faults caused by \( j^{th} \) newly developed component by time \( t \)
\( m_k(t) \) Mean number of faults removed from \( j^{th} \) newly developed component by time \( t \)
\( m_j(t) \) Mean number of faults removed from \( k^{th} \) newly developed component by time \( t \)
\( m_k(t) \) Mean number of faults removed from \( k^{th} \) newly developed component by time \( t \)
\( p \) Reused components having simple faults
\( q \) Newly developed components having hard faults
\( s \) Newly developed components having complex faults

3.1.1 GENERAL FRAMEWORK OF THE PROPOSED MODEL

Time required for fault removal depends on the complexity of faults. The fault is classified as simple if the time delay between failure observation, isolation and removal is negligible. If there is a time delay between failure observation and isolation, the fault is classified as hard fault. If there is a time delay between failure observation, isolation and removal, the fault is classified as a complex fault. It is assumed that simple faults can be removed instantly as soon as they are observed. Software faults in the newly developed software component can be either hard or complex.

MODELLING THE SIMPLE FAULTS

It is assumed that the faults in the reused components are simple faults, which can be removed instantly as soon as they
are observed. Hence fault removal in reused components is modeled as one-stage processes:

\[
\frac{d}{dt} m_p(t) = b_j(a_i - m_p(t))
\]

(4)

Solving the differential equation (4) under the boundary condition \( m_p(t=0) = 0 \), we get:

\[
m_p(t) = a_i (1 - e^{-b_j t})
\]

(5)

**MODELLING THE HARD FAULTS**

It is assumed that the faults of some newly developed components consume more testing effort when compared with faults of reused component. This means that the testing team will have to spend more time to analyze the cause of the failure and therefore requires greater efforts to remove them. Hence the removal process for such faults is modeled as a two stage process:

\[
\frac{d}{dt} m_{pf}(t) = b_j(a_i - m_{pf}(t))
\]

(6)

\[
\frac{d}{dt} m_{pr}(t) = b_j(m_{pf}(t) - m_{pr}(t))
\]

(7)

The first stage of the two-stage process is given by the equation (6). This stage describes the failure observation process. The second stage of the two-stage process given by equation (7) describes the delayed fault removal process. Solving, the above differential equations under the boundary condition, \( t = 0, m_{pf}(t=0) = 0 \) and \( m_{pr}(t=0) = 0 \), we get:

\[
m_{pr}(t) = a_j \left[ 1 - \left( 1 + b_j t \right) e^{-b_j t} \right]
\]

(8)

**MODELLING THE COMPLEX FAULTS**

Some of the newly developed components have complex faults. These faults can require more effort for removal after isolation. Hence they need to be modeled with greater time lag between failure observation and removal. The third stage added below to the model serves the purpose.

\[
\frac{d}{dt} m_{kf}(t) = b_k(a_i - m_{kf}(t))
\]

(9)

\[
\frac{d}{dt} m_{ki}(t) = b_k(m_{kf}(t) - m_{ki}(t))
\]

(10)

\[
\frac{d}{dt} m_{kr}(t) = b_k(m_{ki}(t) - m_{kr}(t))
\]

(11)

The first stage of the three-stage process is given by the equation (9). This stage describes the failure observation process. The second stage given by equation (10) describes the fault isolation process. The third stage given by equation (11) describes the fault removal process. Solving, the above differential equations under the boundary condition, \( t = 0, m_{kf}(t = 0) = 0, m_{ki}(t = 0) = 0 \) and \( m_{kr}(t = 0) = 0 \), we get:

\[
m_{kr}(t) = a_k \left[ 1 - \left( 1 + b_k t + \frac{b_k^2 t^2}{2} \right) e^{-b_k t} \right]
\]

**Modeling Total Fault Removal Phenomenon**

The proposed model is the sum of ‘p’ reused and ‘q’ & ‘s’ newly developed components. Equations (5), (8) and (12) are mean value functions of respective NHPP [6]. Thus, the mean value function of superimposed NHPP is:

\[
m(t) = \sum_{i=1}^{p} m_p(t) + \sum_{j=p+1}^{p+q} m_j(t) + \sum_{k=p+q+1}^{p+q+s} m_k(t)
\]

(12)

\[
m(t) = \sum_{i=1}^{p} a_i (1 - e^{-b_i t}) + \sum_{j=p+1}^{p+q} \left[ 1 - \left( 1 + b_j t \right) e^{-b_j t} \right] + \sum_{k=p+q+1}^{p+q+s} \left[ 1 - \left( 1 + b_k t + \frac{b_k^2 t^2}{2} \right) e^{-b_k t} \right]
\]

(13)

3.2 SOME PARTICULAR CASES OF THE PROPOSED MODEL AND THEIR NEURAL-NETWORK ARCHITECTURES

In this subsection, we discuss some particular cases of the proposed model. Further, we build the neural-network for different cases. Network-network architecture for the first case is given in figure 2. Figure 3 shows the neural-network architecture for the other four cases with different initial weights and activation functions in hidden layer for each case.

The activation function for the unit in output layer is each of the case is defined as

\[ g(x) = x \]

We assume that there is no bias in units of hidden layer and output layer of the neural-network in all the cases.

**Model 1-1-1 (Case 1)**

In this case, software system consists of one reused (i.e. \( p=1 \)) and two newly developed components (one of which contains faults represented by a two stage fault removal phenomenon while other contains faults represented by a three stage fault removal phenomenon i.e. \( p=1, q = 1 \) and \( s = 1 \) respectively). Accordingly equation (13) can be rewritten as

\[
m(t) = a_1 (1 - e^{-b_1 t}) + a_2 \left[ 1 - \left( 1 + b_2 t + \frac{b_2^2 t^2}{2} \right) e^{-b_2 t} \right] + a_3 \left[ 1 - \left( 1 + b_3 t + \frac{b_3^2 t^2}{2} \right) e^{-b_3 t} \right]
\]

(14)

where \( a_1 + a_2 + a_3 = a \)
Assessing Software Reliability Growth with Different Combinations For Distributed Environment Using ANN Approach

Figure 2 shows the network architecture of Model 1-1-1. The $j^{th}$ unit in hidden layer will have the activation function $f_j(x)$. The activation functions for the units in hidden layer are defined as:

$$f_1(x) = 1 - e^{-x},$$
$$f_2(x) = 1 - (1 + x)e^{-x} \quad \text{and}$$
$$f_3(x) = 1 - (1 + x + \frac{x^2}{2})e^{-x}$$

$w_{1j}, w_{2j} (j=1, 2 \text{ and } 3)$ are the weights of the network. Note that $w_{1j} (j=1, 2 \text{ and } 3)$ are the fault detection rates for simple, hard and complex faults respectively. $w_{2j} (j=1, 2 \text{ and } 3)$ are the weights or proportion of total fault content in the software of simple, hard and complex fault types respectively.

Output from the single unit of output layer is

$$y(t) = g(y_m(t)) = g(w_{11}(1-e^{-w_1t}) + w_{12}(1-(1+w_1t)e^{-w_2t}) + w_{13}(1-(1+w_1t)e^{-w_3t}))$$

$$= w_{21}(1-e^{-w_1t}) + w_{22}(1-(1+w_1t)e^{-w_2t}) + w_{23}(1-(1+w_1t)e^{-w_3t})$$

$$= w_{21}(1-e^{-w_1t}) + w_{22}(1-(1+w_1t)e^{-w_2t}) + w_{23}(1-(1+w_1t)e^{-w_3t})$$

Using $w_{11} = b_1, w_{21} = a_1, w_{12} = b_2, w_{22} = a_2, w_{13} = b_3$ and $w_{23} = a_3$, equation (15) is same as equation (14), which is the mean value function for Model 1-1-1.

\[ m(t) = \sum_{i=1}^{2} a_i (1 - e^{-b_i t}) + a_3 \left[ 1 - \left( 1 + b_3 t \right) e^{-b_3 t} \right] + \frac{a_4}{2} \left[ 1 - \left( 1 + b_4 t + \frac{b_4^2 t^2}{2} \right) e^{-b_4 t} \right] \] (16)

where $a_1 + a_2 + a_3 + a_4 = a$

Figure 3 shows the network architecture of Model 2-1-1. The $j^{th}$ unit in hidden layer will have the activation function $f_j(x)$. The activation functions for the units in hidden layer are defined as:

$$f_1(x) = 1 - e^{-x},$$
$$f_2(x) = 1 - e^{-x},$$
$$f_3(x) = 1 - (1 + x)e^{-x} \quad \text{and}$$
$$f_4(x) = 1 - (1 + x + \frac{x^2}{2})e^{-x}$$

$w_{1j}, w_{2j} (j=1, 2, 3 \text{ and } 4)$ are the weights of the network. Note that $w_{1j} (j=1, 2)$ are the fault detection rates for simple faults and $w_{1j} (j=3 \text{ and } 4)$ are the fault detection rates for hard and complex faults respectively. $w_{2j} (j=1 \text{ and } 2)$ are the weights or proportion of total fault content in the software of simple faults and $w_{2j} (j=3 \text{ and } 4)$ are the weights or proportion of total fault content in the software of hard and complex fault types respectively.

Output from the single unit of output layer is

$$y(t) = g(y_m(t)) = g(w_{11}(1-e^{-w_1t}) + w_{21}(1-(1+w_1t)e^{-w_2t}) + w_{22}(1-(1+w_1t)e^{-w_3t})) + w_{23}(1-(1+w_1t)e^{-w_4t}$$

$$= w_{21}(1-e^{-w_1t}) + w_{22}(1-(1+w_1t)e^{-w_2t}) + w_{23}(1-(1+w_1t)e^{-w_3t}) + w_{24}(1-(1+w_1t)e^{-w_4t})$$

Using $w_{11} = b_1, w_{21} = a_1, w_{12} = b_2, w_{22} = a_2, w_{13} = b_3, w_{23} = a_3, w_{14} = b_4$, and $w_{24} = a_4$ equation (17) is same as equation (16), which is the mean value function for Model 2-1-1.

\[ m(t) = \sum_{i=1}^{2} a_i (1 - e^{-b_i t}) + a_3 \left[ 1 - \left( 1 + b_3 t \right) e^{-b_3 t} \right] + \frac{a_4}{2} \left[ 1 - \left( 1 + b_4 t + \frac{b_4^2 t^2}{2} \right) e^{-b_4 t} \right] \] (16)

where $a_1 + a_2 + a_3 + a_4 = a$
Model 2-2-0 (Case 3)
Software system consists of two reused and two newly developed sub-systems i.e. \( p = 2 \) and \( q = 2 \) respectively. Accordingly equation (13) can be rewritten as

\[
m(t) = \sum_{i=1}^{2} a_i (1 - e^{-b_i t}) + \sum_{j=1}^{4} a_j [1 - \left(1 + b_j t\right) e^{-b_j t}]
\]  

(18)

where \( a_1 + a_2 + a_3 + a_4 = a \)

The \( j^{th} \) unit in hidden layer will have the activation function \( f_j(x) \). The activation functions for the units in hidden layer are defined as:

\[
f_1(x) = 1 - e^{-x}, \quad f_2(x) = 1 - e^{-x}, \quad f_3(x) = 1 - (1 + x) e^{-x} \quad \text{and} \quad f_4(x) = 1 - (1 + x) e^{-x}
\]

\( w_1, w_2 \) (j=1, 2, 3 and 4) are the weights of the network. Note that \( w_1 \) (j=1, 2) are the fault detection rates for simple faults and \( w_1 \) (j=3 and 4) are the fault detection rates for hard. \( w_2 \) (j=1 and 2) are the weights or proportion of total fault content in the software of simple faults and \( w_2 \) (j=3 and 4) are the weights or proportion of total fault content in the software of hard faults.

Output from the single unit of output layer is

\[
y(t) = g(y_m(t)) = g(\sum_{i=1}^{2} w_{2i}(1 - e^{-w_i t}) + \sum_{j=1}^{4} w_{2j} (1 - (1 + w_j t) e^{-w_j t})
\]

(19)

Using \( w_{11} = b_1, w_{21} = a_1, w_{12} = b_2, w_{22} = a_2, w_{13} = b_3, w_{23} = a_3, w_{14} = b_4, \) and \( w_{24} = a_4 \),

equation (19) is same as equation (18), which is the mean value function for Model 2-2-0.

Model 1-2-1 (Case 4)
Software system consists of two reused and two newly developed sub-systems i.e. \( p = 1, q = 2 \) and \( s=1 \) respectively. Accordingly equation (13) can be rewritten as

\[
m(t) = a_1 (1 - e^{-b_1 t}) + \sum_{j=2}^{3} a_j [1 - \left(1 + b_j t\right) e^{-b_j t}]
\]

(20)

where \( a_1 + a_2 + a_3 = a \)

The \( j^{th} \) unit in hidden layer will have the activation function \( f_j(x) \). The activation functions for the units in hidden layer are defined as:

\[
f_1(x) = 1 - e^{-x}, \quad f_2(x) = 1 - (1 + x) e^{-x} \quad \text{and} \quad f_3(x) = 1 - (1 + x) e^{-x}
\]

\( w_1, w_2 \) (j=1, 2, 3 and 4) are the weights of the network. Note that \( w_1 \) (j=1, 2) are the fault detection rates for simple faults and \( w_1 \) (j=3 and 4) are the fault detection rates for hard. \( w_2 \) (j=1 and 2) are the weights or proportion of total fault content in the software of simple faults and \( w_2 \) (j=3 and 4) are the weights or proportion of total fault content in the software of hard faults.

Output from the single unit of output layer is
Assessing Software Reliability Growth with Different Combinations For Distributed Environment Using ANN Approach

\begin{equation}
y(t) = g(y_n(t)) = g(w_{j1}(1-e^{-\eta_1}) + w_{21}(1- \eta_2)e^{-\eta_2}) \nonumber \\
+ w_{31}(1- (1+w_1d)e^{-\eta_1}) + w_{41}(1- (1+w_1d + \frac{w_{12}^2 \eta_1}{2})e^{-\eta_1}) \nonumber \\
= w_{j1}(1-e^{-\eta_1}) + w_{21}(1- (1+w_1d)e^{-\eta_2}) \nonumber \\
+ w_{31}(1- (1+w_1d)e^{-\eta_1}) + w_{41}(1- (1+w_1d + \frac{w_{12}^2 \eta_1}{2})e^{-\eta_1}) \nonumber \\
\text{(21)}
\end{equation}

Using \( w_{11} = b_1, w_{21} = a_1, w_{12} = b_2, w_{22} = a_2, \)
\( w_{13} = b_3, w_{23} = a_3, w_{14} = b_4, \) and \( w_{24} = a_4 \).

equation (21) is same as equation (20), which is the mean value function for Model 1-2-1.

**Model 1-1-2 (Case 5)**

Software system consists of two reused and two newly developed sub-systems i.e. \( p = 1, q = 1 \) and \( s=2 \) respectively. Accordingly equation (13) can be written as:

\[ m(t) = a_1(1 - e^{-b_1t}) + \sum_{k=2}^{4} a_k \left[ 1 - \left\{ 1 + b_k t \frac{b_k^2 \eta}{2} \right\} e^{-b_k t} \right] + \]

\[ \nonumber \sum_{k=3}^{4} a_k \left[ 1 - \left\{ 1 + b_k t \frac{b_k^2 \eta}{2} \right\} e^{-b_k t} \right] \text{(22)} \]

where \( a_1 + a_2 + a_3 + a_4 = a \)

The \( j^{th} \) unit in hidden layer will have the activation function \( f_j(x) \). The activation functions for the units in hidden layer are defined as:

\( f_1(x) = 1 - e^{-x}, \)
\( f_2(x) = 1 - (1 + x)e^{-x}, \)
\( f_3(x) = 1 - (1 + x + \frac{x^2}{2})e^{-x}, \) and
\( f_4(x) = 1 - (1 + x + \frac{x^2}{2})e^{-x}, \)

\( w_{1j}, w_{2j} (j=1, 2, 3 \text{ and } 4) \) are the weights of the network.

Note that \( w_{1j} \) (\( j=1 \) and 2) are the fault detection rates for simple and hard faults respectively and \( w_{1j} \) (\( j=3 \) and 4) are the fault detection rates for complex faults. \( w_{2j} \) (\( j=1 \) and 2) are the weights or proportion of total fault content in the software of simple and hard faults respectively and \( w_{2j} \) (\( j=3 \) and 4) are the weights or proportion of total fault content in the software of complex fault types.

Output from the single unit of output layer is

\begin{equation}
y(t) = g(y_n(t)) = g(w_{j1}(1-e^{-\eta_1}) + w_{21}(1- \eta_2)e^{-\eta_2}) \nonumber \\
+ w_{31}(1- (1+w_1d)e^{-\eta_1}) + w_{41}(1- (1+w_1d + \frac{w_{12}^2 \eta_1}{2})e^{-\eta_1}) \nonumber \\
= w_{j1}(1-e^{-\eta_1}) + w_{21}(1- (1+w_1d)e^{-\eta_2}) \nonumber \\
+ w_{31}(1- (1+w_1d)e^{-\eta_1}) + w_{41}(1- (1+w_1d + \frac{w_{12}^2 \eta_1}{2})e^{-\eta_1}) \nonumber \\
\text{(23)}
\end{equation}

Using \( w_{11} = b_1, w_{21} = a_1, w_{12} = b_2, w_{22} = a_2, \)
\( w_{13} = b_3, w_{23} = a_3, w_{14} = b_4, \) and \( w_{24} = a_4 \).

equation (23) is same as equation (22), which is the mean value function for Model 1-1-2.

4 **PARAMETER ESTIMATION**

The success of mathematical modeling approach to reliability evaluation depends heavily upon quality of failure data collected, software development environment, software personnel, and so on. Neural Network approach is based on building a network of neurons with weights initialized which are changed during training using back propagation method to minimize the mean squared error. The parameters of the proposed model (for different cases) are estimated based upon these data. We had written the code for learning and training the neural network in “C” language. The data set is passed to this code to estimate the parameters for different cases of the proposed model.

5 **MODEL VALIDATION**

To assess the performance of the proposed model, we have carried out the parameter estimation on two real software failure datasets.

**Data set 1(DS-1)**

The first data set (DS-1) had been collected during 38 weeks and 231 faults were detected during testing. This data is cited from Misra [14].

**Data set 2(DS-2)**

The second data set (DS-2) had been collected during 21 weeks of testing and 26 faults were detected during testing. This data is cited from Pham [16].

5.1 **Comparison Criteria for SRGM**

The performance of SRGM are judged by their ability to fit the past software fault data (Goodness-of-Fit).

**Goodness of Fit criteria**

The term goodness of fit denotes the question of “How good does a model fit to the data”?

**a. The Mean Square Fitting Error (MSE):**

The model under comparison is used to simulate the fault data, the difference between the expected values, \( \hat{m}(t_j) \) and the observed data \( y_j \), is measured by MSE as follows.
where \( k \) is the number of observations. The lower MSE indicates less fitting error, thus better goodness of fit [6].

b. Bias:
Prediction error (PE) is the difference between the actual and the estimated number of faults. The average of PEs is known as bias. Lower the value of Bias better is the goodness of fit [17].

c. Variation:
The standard deviation of prediction error is known as variation.

\[
\text{Variation} = \sqrt{\frac{1}{N-1} \sum (PE - \text{Bias})^2}
\]

Lower the value of Variation better is the goodness of fit [17].

d. Root Mean Square Prediction Error:
It is a measure of closeness with which a model predicts the observation.

\[
\text{RMSPE} = \sqrt{\text{Bias}^2 + \text{Variation}^2}
\]

Lower the value of Root Mean Square Prediction Error better is the goodness of fit [17].

6 DATA ANALYSIS AND MODEL COMPARISON

Parameters of neural-network based proposed model (different cases) are estimated. To judge the accuracy of the proposed model we had used MSE, Bias, Variation and RMSPE as the performance measures. The estimated values of parameters of proposed model for different cases are worked out. For comparison, the comparison criteria are applied. The estimation results are provided in Table 2 and Table 3 while the comparison criteria results are shown in Table 2 and Table 4.

The software system built in distributed environment can be considered to have three types of components on the basis of type of faults. It can be observed from Table 2 and 4 that Model 1-1-1 gives better result in comparison to other models for software developed in distributed environment.

### Table 1 For DS-1 (Parameter Estimation)

<table>
<thead>
<tr>
<th></th>
<th>Model 1-1-1 (equation 14)</th>
<th>Model 2-1-1 (equation 16)</th>
<th>Model 2-2-0 (equation 18)</th>
<th>Model 1-2-1 (equation 20)</th>
<th>Model 1-1-2 (equation 22)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>539.09</td>
<td>425.922</td>
<td>459.867</td>
<td>603.186</td>
<td>597.846</td>
</tr>
<tr>
<td>( b_1 )</td>
<td>0.631</td>
<td>0.929</td>
<td>0.918</td>
<td>0.621</td>
<td>0.625</td>
</tr>
<tr>
<td>( b_2 )</td>
<td>0.332</td>
<td>0.889</td>
<td>0.889</td>
<td>0.377</td>
<td>0.531</td>
</tr>
<tr>
<td>( b_3 )</td>
<td>0.107</td>
<td>0.830</td>
<td>0.230</td>
<td>0.460</td>
<td>0.108</td>
</tr>
<tr>
<td>( b_4 )</td>
<td>---</td>
<td>0.101</td>
<td>0.930</td>
<td>0.101</td>
<td>0.143</td>
</tr>
<tr>
<td>( p_1 )</td>
<td>0.857</td>
<td>0.216</td>
<td>0.221</td>
<td>0.766</td>
<td>0.773</td>
</tr>
<tr>
<td>( p_2 )</td>
<td>0.099</td>
<td>0.560</td>
<td>0.502</td>
<td>0.102</td>
<td>0.151</td>
</tr>
<tr>
<td>( p_3 )</td>
<td>0.044</td>
<td>0.217</td>
<td>0.076</td>
<td>0.127</td>
<td>0.040</td>
</tr>
<tr>
<td>( p_4 )</td>
<td>---</td>
<td>0.007</td>
<td>0.201</td>
<td>0.005</td>
<td>0.037</td>
</tr>
</tbody>
</table>

### Table 2 For DS-1 (Comparison Criterion)

<table>
<thead>
<tr>
<th></th>
<th>MSE</th>
<th>Bias</th>
<th>Variation</th>
<th>RMSPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model 1-1-1 (equation 14)</td>
<td>20.144</td>
<td>-1.063</td>
<td>4.419</td>
<td>4.545</td>
</tr>
</tbody>
</table>

### Table 3 For DS-2 (Parameter Estimation)

<table>
<thead>
<tr>
<th></th>
<th>Model 1-1-1 (equation 14)</th>
<th>Model 2-1-1 (equation 16)</th>
<th>Model 2-2-0 (equation 18)</th>
<th>Model 1-2-1 (equation 20)</th>
<th>Model 1-1-2 (equation 22)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>60.462</td>
<td>61.206</td>
<td>65.402</td>
<td>60.202</td>
<td>58.167</td>
</tr>
<tr>
<td>( b_1 )</td>
<td>0.895</td>
<td>0.616</td>
<td>0.621</td>
<td>0.839</td>
<td>0.755</td>
</tr>
<tr>
<td>( b_2 )</td>
<td>0.874</td>
<td>0.675</td>
<td>0.611</td>
<td>0.306</td>
<td>0.185</td>
</tr>
<tr>
<td>( b_3 )</td>
<td>0.424</td>
<td>0.149</td>
<td>0.179</td>
<td>0.497</td>
<td>0.102</td>
</tr>
<tr>
<td>( b_4 )</td>
<td>---</td>
<td>0.102</td>
<td>0.252</td>
<td>0.206</td>
<td>0.207</td>
</tr>
<tr>
<td>( p_1 )</td>
<td>0.599</td>
<td>0.406</td>
<td>0.447</td>
<td>0.734</td>
<td>0.841</td>
</tr>
<tr>
<td>( p_2 )</td>
<td>0.344</td>
<td>0.497</td>
<td>0.426</td>
<td>0.085</td>
<td>0.067</td>
</tr>
<tr>
<td>( p_3 )</td>
<td>0.056</td>
<td>0.054</td>
<td>0.057</td>
<td>0.136</td>
<td>0.045</td>
</tr>
<tr>
<td>( p_4 )</td>
<td>---</td>
<td>0.043</td>
<td>0.071</td>
<td>0.045</td>
<td>0.047</td>
</tr>
</tbody>
</table>

### Table 4 For DS-2 (Comparison Criterion)

<table>
<thead>
<tr>
<th></th>
<th>MSE</th>
<th>Bias</th>
<th>Variation</th>
<th>RMSPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model 1-1-1 (equation 14)</td>
<td>5.527</td>
<td>1.283</td>
<td>2.019</td>
<td>2.392</td>
</tr>
<tr>
<td>Model 2-1-1 (equation 15)</td>
<td>7.408</td>
<td>1.856</td>
<td>2.040</td>
<td>2.758</td>
</tr>
<tr>
<td>Model 2-2-0 (equation 16)</td>
<td>6.981</td>
<td>1.765</td>
<td>2.015</td>
<td>2.679</td>
</tr>
<tr>
<td>Model 1-2-1 (equation 17)</td>
<td>7.877</td>
<td>1.852</td>
<td>2.161</td>
<td>2.846</td>
</tr>
<tr>
<td>Model 1-1-2 (equation 18)</td>
<td>8.777</td>
<td>2.181</td>
<td>2.055</td>
<td>2.996</td>
</tr>
</tbody>
</table>

Figure 4 and 5 shows goodness-of-fit curves for different cases of the proposed model for DS-1 and DS-2.

Goodness-of-Fit Curves For The Two Data Sets
Computing has now reached the state of distributed computing. Several SRGM have been developed in the literature to estimate the fault content and fault removal rate per fault in software. Recently, artificial neural-network approach has been applied in estimating and predicting software reliability growth phenomenon. Neural-networks may handle numerous factors and approximate any non-linear functions. In this paper a neural-network based model is proposed for software built in distributed environment. Different cases of the proposed model have been discussed. These cases have been validated, evaluated and compared by applying two actual data sets cited from real software development projects. Results show that the Model 1-1-1 gives better result in comparison to other cases in distributed environment.

REFERENCES

**Continued from Page No. 376**

**SOFTWARE COST OR PRODUCT??**

Many look upon software testing as a cost. While it is true that software testing does cost money, in many cases significant amounts of money, it is also an activity that helps an organization to avoid costly failures further on in the development process. Most understand this relationship; software testing is spending money to save money. What many do not also realize is that software testing also produces valuable assets for the organization because of these reasons.

**The Products of Software Testing**

Software testing is a process that results in products. These include the following:

1. The test design process produces a series of test cases.
2. The test execution process produces a list of software anomalies.
3. The problem identification process produces bug reports.

Organizations think of testing as an activity as opposed to a product. It is seen in terms of costs and savings, as the activity costs money but finding bugs early saves money. By thinking of testing as producing products, and considering these products as assets, an organization’s approach to testing can be significantly enhanced. Consider the following points that come out of this way of thinking:

- Test cases have a definite value and can depreciate over time as the underlying application changes.
- Well-written test cases consolidate the intellectual property of your team members and retain that knowledge in the company as staff members may come and go.
- Well-automated tests can be re-used over and over again, becoming assets that produce profits for the company.

So, by taking into consideration the products of software testing, the expenses incurred for software testing can be looked upon in a different light. While it is an expense to avoid later costs, it also produces reusable assets that should be considered as both helping to offset the costs of software testing while strengthening the organization by codifying its intellectual property.

**CONCLUSION**

In this, we discussed the basic principles of black box testing and white box testing. We have surveyed some of the strategies supporting both paradigms, and have discussed their pros and cons. Because both paradigms are complementary rather than contradictory, we have proposed a testing process, that starts with the development of a black test suite early in the life cycle, and that is complemented with a white test suite late in the life cycle. Different independent versions of same software are used to compare to each other for testing in this method.

Testing is a process of technical investigation, performed on behalf of stakeholders, that is intended to reveal quality-related information about the product with respect to the context in which it is intended to operate. A primary purpose for testing is to detect software failures so that defects may be uncovered and corrected. Software Testing is the process of executing software in a controlled manner; in order to answer the question “Does this software behave as specified?” Software Testing is the process used to help identify the correctness, completeness and quality of developed computer software. Software testing accounts for a large percentage of effort in the software development process, but we have only recently begun to understand the subtleties of systematic planning, execution and control.

**FUTURE SCOPE**

This paper attempts to predict how testing can become more effective and efficient, so that it can keep pace with, and support the changing trends in software development. This paper helps the tester in selecting the proper test method that helps in conducting “true” testing the selection of the proper method of testing is driven by the environment, the situation and most importantly by the objectives. As a general rule, no one method alone is sufficient. A combination of methods and techniques is usually necessary to develop a good set of test cases against which the software can be evaluated. This paper investigates how testing will need to change to accommodate these trends and become a business led activity.

**REFERENCES**