Using an RBF Neural Network to Locate Program Bugs

W. Eric Wong, Yan Shi, Yu Qi
Department of Computer Science
University of Texas at Dallas
{ewong, yxs055100, yxq014100}@utdallas.edu

Richard Golden
School of Behavioral and Brain Sciences
University of Texas at Dallas
golden@utdallas.edu

Abstract

We propose an RBF (radial basis function) neural network-based fault localization method to help programmers locate bugs in a more effective way. An RBF neural network with a three-layer feed-forward structure is employed to learn the relationship between the statement coverage of a test case and its corresponding execution result. The trained network is then given as input a set of *virtual* test cases, each covering only a single statement. The output of the network for each test case is considered to be the *suspiciousness* of the corresponding statement; a statement with a higher suspiciousness has a higher likelihood of containing a bug. The set of statements ranked in descending order by their suspiciousness are then examined by programmers one by one until a bug is located. Three case studies on different programs (space, grep and make) were conducted with each faulty version having exactly one bug. An additional program gcc was also used to demonstrate the concept of extending the proposed method to programs with multiple bugs. Our experimental data suggest that an RBF neural network-based fault localization method is more effective in locating a program bug (by examining less code before the first faulty statement containing the bug is identified) than another popular method, Tarantula, which also uses the coverage and execution results to compute the suspiciousness of each statement.

Keywords: fault localization, program debugging, RBF (radial basis function) neural network, suspiciousness of code, successful test, failed test, *EXAM* score

1. Introduction

No matter how much effort we spent on developing a computer program,[1] it will still contain bugs. In fact, the more complicated the program, the more likely it contains bugs. To remove program bugs, we must first find where the bugs are. Fault localization is a very expensive process. It can be divided into two major parts. The first part is to use a method to identify suspicious code (executable statements[2] in our case) that may contain program bugs. The second part is for programmers to actually examine the identified code to decide whether it indeed contains the bugs. All the fault localization methods referenced in this paper, including our RBF (radial basis function) neural network-based method (hereafter, referred to as RBFNN), focus on the first part.

Neural networks have attracted a great deal of attention from researchers because they have many advantages over other models. For example, they have the ability to learn. Given sample data, a neural network can learn rules from these sample data with or without a teacher. Neural networks are fault tolerant; because the information is distributed among the weights on the connections, a few faults in the network have little impact on the model. They have the capability to adapt their synaptic weights to changes in the surrounding environment. That is, a neural network trained to operate in a specific environment can be easily retrained to deal with minor change in the operating environmental conditions.

[1]In this paper, we use "programs," "applications" and "software" interchangeably. We also use "bugs," "faults," and "defects" interchangeably. In addition, "a statement is covered by a test case" and "a statement is executed by a test case" are used interchangeably.

[2]All the comments, blank lines, non-executable statements (e.g., function and variable declarations) are excluded for analysis. When there is no ambiguity, we refer to "executable statements" as simply "statements" from this point on.

Neural networks have been successfully applied to many fields, such as pattern recognition [7], system identification [4], intelligent control [23], and software engineering areas including risk analysis [24], cost estimation [31], reliability estimation [30], and reusability characterization [3]. However, they have not been applied to help programmers find bugs except for our previous study [38] which uses a back-propagation (BP) neural network-based method for fault localization. In this paper we propose to use an RBF neural network-based fault localization method because RBF networks have several advantages over BP networks, including a faster learning rate and a resistance to problems such as paralysis and local minima [16,33].

A typical RBF neural network has a three-layer feed-forward structure that can be trained to learn the input-output relationship from a set of data. In this paper, the input is the statement coverage of a test case and the output is the corresponding execution result (success or failure). After the network is trained, the coverage of a *virtual* test case with only one statement covered[1] is used as an input to compute the *suspiciousness* of the corresponding statement in terms of its likelihood of containing bugs. The larger the output is, the more suspicious the statement. Statements are then ranked in descending order based on their likelihood of containing bugs. Programmers examine these statements from the top of the rank one by one until the first statement containing the bugs is identified. Three programs (space, grep and make) with faulty versions each having exactly one bug and a program (gcc) with multiple bugs are used to demonstrate the effectiveness of our RBFNN method (see Sections 5 and 6). Data from our experiments suggest that RBFNN is more effective than Tarantula, a popular fault localization method, which uses the same input (coverage and execution results) to rank statement suspiciousness.

There are three novel aspects to our work. First, we introduce a method for representing test cases, statement coverage, execution results within a modified RBF neural network formalism. Moreover, the formulation of the problem in terms of training an artificial neural network with example test cases and execution results, and then testing with virtual test cases is novel. Second, we developed a novel algorithm (Figure 5) to simultaneously estimate the number of hidden neurons and their receptive field centers. Third, instead of using the traditional Euclidean distance which has been proved to be inappropriate in the fault localization context (Section 3.3), a weighted bit-comparison based distance is defined to measure the distance between the statement coverage vectors of two test cases. Such distance is used to 1) estimate the number of hidden neurons and their receptive field centers, and 2) compute the output of each hidden neuron.

2. An Overview of RBF Neural Networks

In this section, we first present a general description of neural networks followed by a more specific discussion on the RBF neural networks.

2.1 Neural networks

Traditionally, a neural network has referred to a network of biological neurons. The modern definition of this term is an artificial construct whose behavior is based on that of a network of artificial neurons. These neurons are connected together with weighted connections following a certain structure. Each neuron has an activation function that describes the relationship between the input and the output of the neuron [8]. The data can be processed in parallel by different neurons and distributed on the weights of the connections between neurons. Different neural network models have been developed including BP neural networks [8], RBF neural networks [9], self-organizing map (SOM) neural networks [11], and adaptive resonance theory (ART) neural networks [9].

A very important attribute of a neural network is that it can learn from experience. Such learning is normally accomplished through an adaptive procedure, known as a learning algorithm. These algorithms can be divided into two categories: *supervised* and *unsupervised* [33]. Each network learning algorithm has certain strengths and weaknesses in the areas of reliability, performance, and generality; however, none has a clear advantage over another.

In fault localization, the output of a given input can be defined as a binary value of 0 or 1, where 1 represents a program failure on this input and 0 represents a successful execution. With this definition, the output of each input is known because we know exactly whether the corresponding program execution fails or succeeds. Moreover, two similar inputs can produce different outputs because the program execution may fail on one input but succeed on another input. This makes unsupervised learning algorithms inappropriate for our study because those algorithms adjust network weights so that similar inputs produce similar outputs. Therefore, neural networks using supervised learning algorithms are better candidates for solving the fault localization problem. Although BP networks are the widely used networks for supervised learning, RBF networks (whose output layer weights are trained in a supervised way) are even better in our case because they can learn much faster than BP networks and do not suffer from pathologies like paralysis and local minima problem as BP networks [16,33].

2.2 RBF neural networks

A radial basis function (RBF) is a real-valued function whose value depends only on the "distance" from its receptive field center μ to the input $\mathbf{x}$. It is a strictly positive radially symmetric function, where the center has the unique maximum and the value drops off rapidly to zero away from the center. When the distance between $\mathbf{x}$ and μ (denoted as $||\mathbf{x}-\mu||$) is smaller than the receptive field width σ, the function has an appreciable value.

A typical RBF neural network has a three-layer feed-forward structure. The first layer is the input layer which serves as an input distributor to the hidden layer by passing inputs to the hidden layer without changing their values. The second layer is the hidden layer where all neurons simultaneously receive the n-dimensional real-valued input vector $\mathbf{x}$. Each neuron in this layer uses an RBF as the activation function. A commonly used RBF is the Gaussian basis function [9]

$$R_j(\mathbf{x}) = \exp\left(-\frac{\|\mathbf{x}-\boldsymbol{\mu}_j\|^2}{2\sigma_j^2}\right) \tag{1}$$

where μ_j and σ_j are the mean (namely, the center) and the standard deviation (namely, the width) of the receptive field of the jth hidden layer neuron, and $R_j(\mathbf{x})$ is the corresponding activation function. Usually the distance in Equation (1) is the Euclidean distance between $\mathbf{x}$ and μ, but in this paper we use a weighted bit-comparison based distance. To make a distinction, hereafter we use $||\mathbf{x}-\mu||$ to represent a *generic* distance, $||\mathbf{x}-\mu||_E$ for the *Euclidean* distance, and $||\mathbf{x}-\mu||_{WBC}$ (Equation (8) in Section 3.3) for the weighted bit-comparison based distance. The third layer is the output layer. The output can be expressed as $\mathbf{y} = [y_1, y_2, \ldots, y_k]$ with y_i as the output of the ith neuron given by

$$y_i = \sum_{j=1}^{h} w_{ji} R_j(\mathbf{x}) \quad \text{for } i = 1, 2, \ldots, k \tag{2}$$

where h is the number of neurons in the hidden layer and w_{ji} is the weight associated with the link connecting the jth hidden layer neuron and the ith output layer neuron.

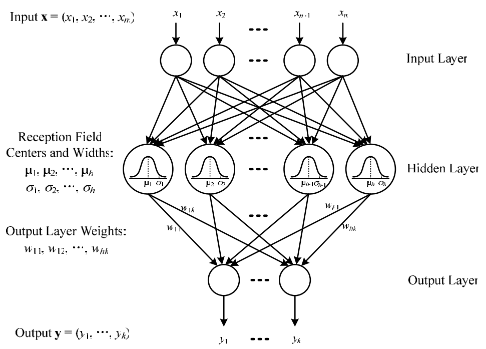

Input $\mathbf{x} = (x_1, x_2, \cdots, x_n)$
x_1
x_2
x_{n-1}
x_n
Input Layer
Reception Field Centers and Widths:
$\mu_1, \mu_2, \cdots, \mu_h$
$\sigma_1, \sigma_2, \cdots, \sigma_h$
Hidden Layer
w_{1k}
w_{11}
w_{h1}
w_{hk}
Output Layer Weights:
$w_{11}, w_{12}, \cdots, w_{hk}$
Output Layer
Output $\mathbf{y} = (y_1, \cdots, y_k)$
y_1
y_k

Figure 1. A sample three-layer RBF neural network

An RBF network implements a mapping from the m dimensional real-valued input space to the k dimensional real-valued output space. In between, there is a layer of hidden-layer space. The transformation from the input space to the hidden-layer space is nonlinear, whereas the transformation from the hidden-layer space to the output space is linear [11]. Figure 1 shows an RBF network with m neurons in the input layer, h neurons in the hidden layer, and k neurons in the output layer. The parameters that need to be trained are the centers (i.e., $\mu_1, \mu_2, \ldots, \mu_h$) and widths (i.e., $\sigma_1, \sigma_2, \ldots, \sigma_h$) of the receptive fields of hidden layer neurons, and the output layer weights. Many methods have been proposed to train these parameters [9]. Section 3.2 explains how they are trained in our study.

3. Our Method

In this section, we first explain the use of an RBF neural network to compute the suspiciousness of each statement in a program P for effective fault localization, followed by a two-stage training of such a network, and then provide the definition of a weighted bit-comparison based distance for computing the output of an RBF network.

3.1 Fault localization using an RBF neural network

Suppose we have a program P with m statements. Suppose also P is executed on n test cases. Let t_i be the i^{th} test case executed on P, $\mathbf{c}_{t_i}$ and r_{t_i} be the coverage vector and the execution result (success or failure) of t_i, respectively, and s_j be the jth statement of P. The vector $\mathbf{c}_{t_i}$ tells how the program P is covered by test t_i. In this paper, such coverage is reported in terms of which statements[3] in P are executed by t_i. We have $\mathbf{c}_{t_i} = \left[(\mathbf{c}_{t_i})_1, (\mathbf{c}_{t_i})_2, \cdots, (\mathbf{c}_{t_i})_m\right]$ where

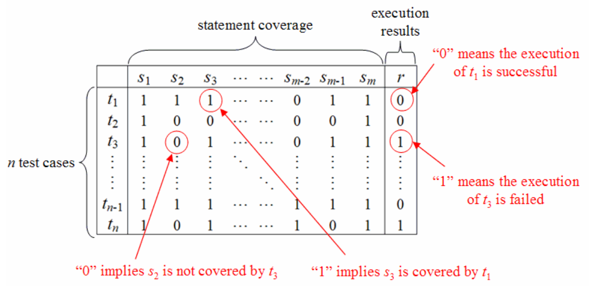
$$(\mathbf{c}_{t_i})_j = \begin{cases} 0, & \text{if statement } s_j \text{ is not covered by test } t_i \\ 1, & \text{if statement } s_j \text{ is covered by test } t_i \end{cases} \quad \text{for } 1 \le j \le m$$

The value of r_{t_i} depends on whether the program execution of t_i succeeds or fails. It has a value 1 if the execution fails and a value 0 if the execution succeeds. Figure 2 (which is best viewed in color) gives an example of the statement coverage with respect to each test case and the corresponding execution result. We observe that statement s_3 is covered by a successful test t_1, and statement s_2 is not covered by a failed test t_3.

We construct an RBF neural network with m input layer neurons (each of which corresponds to one element in a given

[3]To make a fair comparison between the effectiveness of our method and those reported in [14], we collect coverage data in terms of statements. In addition to statement coverage, our method can also be applied to other criteria such as function-entry, basic block, decision, c-use and p-use coverage [40].

$\mathbf{c}_{t_i}$) and one output layer neuron (corresponding to r_{t_i} – the execution result of test t_i). There is also one hidden layer between the input and output layers. The number of neurons in this layer can be determined by using the algorithm in Figure 5 which will be explained in Section 3.2. Each of these neurons uses the Gaussian basis function as the activation function. The receptive field center and width of each hidden layer neuron and the output layer weights are established by training the underlying network.


statement coverage
execution results
"0" means the execution of t_1 is successful
"1" means the execution of t_3 is failed
n test cases
"0" implies s_2 is not covered by t_3
"1" implies s_3 is covered by t_1

$$\begin{array}{c|cccccccc|c} & s_1 & s_2 & s_3 & \cdots & \cdots & s_{m-2} & s_{m-1} & s_m & r \\ \hline t_1 & 1 & 1 & 1 & \cdots & \cdots & 0 & 1 & 1 & 0 \\ t_2 & 1 & 0 & 0 & \cdots & \cdots & 0 & 0 & 1 & 0 \\ t_3 & 1 & 0 & 1 & \cdots & \cdots & 0 & 1 & 1 & 1 \\ \vdots & \vdots & \vdots & \vdots & \ddots & & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & & \ddots & \vdots & \vdots & \vdots & \vdots \\ t_{n-1} & 1 & 1 & 1 & \cdots & \cdots & 1 & 1 & 1 & 0 \\ t_n & 1 & 0 & 1 & \cdots & \cdots & 1 & 0 & 1 & 1 \end{array}$$

Figure 2. Sample coverage data and execution results

Once an RBF network is trained, it provides a good mapping between the input (the coverage vector of a test case) and the output (the corresponding execution result). It can then be used to identify suspicious code of a given program in terms of its likelihood of containing bugs. To do so, we use a set of *virtual* test cases v_1, v_2, …, v_m whose coverage vectors are $\mathbf{c}_{v_1}, \mathbf{c}_{v_2}, \cdots, \mathbf{c}_{v_m}$, where

$$\begin{bmatrix}\mathbf{c}_{v_1}\\ \mathbf{c}_{v_2}\\ \vdots \\ \mathbf{c}_{v_m}\end{bmatrix} = \begin{bmatrix}1&0&\cdots&0\\0&1&\cdots&0\\ \vdots&\vdots&\ddots&\vdots\\0&0&\cdots&1\end{bmatrix} \qquad (3)$$

Note that the execution of test v_j covers only one statement s_j. As reported in [2,35,37], if the execution of a test case fails, program bugs that are responsible for this failure are most likely to be contained in the corresponding execution slice, i.e., in the statements executed by this failed test case.[4,5] Hence, if the execution of v_j fails, the probability that the bugs are contained in s_j is high. This suggests that during the fault localization, we should first examine the statements whose corresponding virtual test case fails. However, the execution results of these virtual tests can rarely be collected in the real world because it is very difficult, if not impossible, to construct such tests.[6] Nevertheless, when the coverage vector $\mathbf{c}_{v_j}$ of a virtual test case v_j is input to the trained neural network, its output $\hat{r}_{v_j}$ is the conditional expectation of whether the execution of v_j fails given $\mathbf{c}_{v_j}$. This implies the larger the value of $\hat{r}_{v_j}$, the more likely that the execution of v_j fails. Together, we have the larger the value of $\hat{r}_{v_j}$, the more likely it is that s_j contains the bug. We can treat $\hat{r}_{v_j}$ as the suspiciousness of s_j in terms of its likelihood of containing the bug. Figures 3 and 4 show the process of using an RBF neural network for fault localization. We summarize this part as follows:

1) Build up a modified RBF neural network with m neurons in the input layer and one in the output layer. Each neuron in the hidden layer uses the Gaussian basis function as its activation function.
2) Determine the number of neurons in the hidden layer h, and the receptive field center and width of each hidden neuron.
3) Use the Moore-Penrose pseudo-inverse to compute the optimal linear mapping from the hidden neurons to the output neuron.
4) Use the coverage vectors $\mathbf{c}_{v_j}, 1 \le j \le m$ defined in Equation (3) as the inputs to the trained network to produce the outputs $\hat{r}_{v_j}$, $1 \le j \le m$.
5) Assign $\hat{r}_{v_j}$ as the suspiciousness of j^{th} statement.
6) Rank statements s_j, $1 \le j \le m$ based on their suspiciousness in descending order and examine the statements one by one from the top until the fault is located.

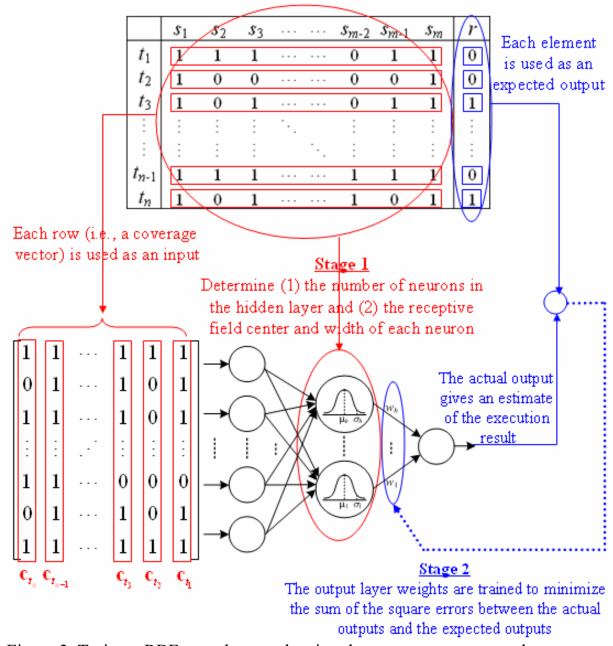

s_1 s_2 s_3 ⋯ ⋯ s_{m-2} s_{m-1} s_m r
t_1 t_2 t_3 t_{n-1} t_n
Each element is used as an expected output
Each row (i.e., a coverage vector) is used as an input
Stage 1
Determine (1) the number of neurons in the hidden layer and (2) the receptive field center and width of each neuron
The actual output gives an estimate of the execution result
Stage 2
The output layer weights are trained to minimize the sum of the square errors between the actual outputs and the expected outputs
$\mathbf{c}_{t_n}$ $\mathbf{c}_{t_{n-1}}$ $\mathbf{c}_{t_3}$ $\mathbf{c}_{t_2}$ $\mathbf{c}_{t_1}$
w_h w_1 μ_h σ_h μ_1 σ_1

Figure 3. Train an RBF neural network using the coverage vectors and program execution results

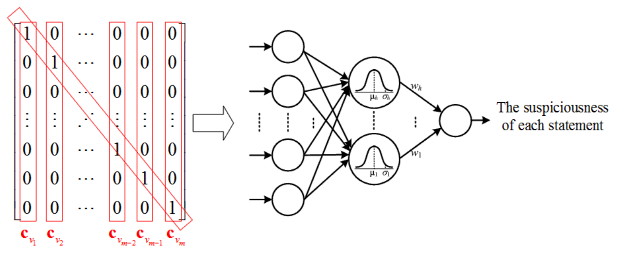

$\mathbf{c}_{v_1}$ $\mathbf{c}_{v_2}$ $\mathbf{c}_{v_{m-2}}$ $\mathbf{c}_{v_{m-1}}$ $\mathbf{c}_{v_m}$
w_h w_1 μ_h σ_h μ_1 σ_1
The suspiciousness of each statement

Figure 4. Compute the suspiciousness of each statement in P using virtual test cases

We want to emphasize that the traditional RBF neural network is modified to better fit our fault localization context. First, in Step 2, we develop a novel algorithm (Figure 5) to simultaneously estimate the number of hidden neurons and their receptive field centers. Second, we define a weighted bit-comparison based distance, instead of using the traditional Euclidean distance, to measure the "distance" between two coverage vectors. Refer to Sections 3.2 and 3.3 for more detailed discussion about these two novelties.

[4]Note that given a program, a statement-based execution slice with respect to a test case is the set of statements in the program executed by this test. Such a slice can be constructed very easily if we know the statement coverage of the test because the corresponding execution slice of the test can be obtained simply by converting the coverage data collected during the testing into another format, i.e., instead of reporting the coverage percentage, it reports which statements are covered.

[5]In some situations, a test case may fail only because a previously executed test did not set up an appropriate execution environment. To account for this, we combine these test cases into a single failed test, with an execution slice consisting of the union of each test case's slice [35,37].

[6]In general, the virtual test cases are not "real" test cases and their coverage vectors are not used as training data for the RBF network.

3.2 Training of the RBF neural network

In this section, we discuss the details of the training procedure as mentioned in Section 3.1. The training of an RBF neural network can be divided into two stages [33]. First, the number of neurons in the hidden layer, the receptive field center $\boldsymbol{\mu}_j$ and width σ_j of each hidden layer neuron should be assigned values. Second, the output layer weights have to be trained.

Many methods have been proposed to determine the receptive field centers. Using standard *k*-means clustering, input data are assigned to *k* clusters, with the center of each cluster taken to be the receptive field center of a hidden layer neuron [6,9,18,32]. Unfortunately, this approach does not provide any guidance as to how many clusters should be used; the number of clusters (and so, the number of receptive field centers) must be chosen arbitrarily. Another disadvantage is that *k*-means is very sensitive to the initial starting values. Its performance will significantly depend on the arbitrarily selected initial receptive field centers.

To overcome these problems, we developed a novel algorithm (as shown in Figure 5) to simultaneously estimate the number of hidden neurons and their receptive field centers. The inputs to this algorithm are the coverage vectors $\{\mathbf{c}_{t_1}, \mathbf{c}_{t_2}, \cdots, \mathbf{c}_{t_n}\}$ and a parameter β ($0 \le \beta < 1$) for controlling the number of field centers. The output is a set of receptive field centers $\{\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \cdots, \boldsymbol{\mu}_h\}$ which is a subset of the input vectors such that $\|\boldsymbol{\mu}_i - \boldsymbol{\mu}_j\|_{\mathrm{WBC}} \ge \beta$ for any *i* and *j*, $i \ne j$, where $\| \;\|_{\mathrm{WBC}}$ is the weighted bit-comparison based distance defined in Section 3.3. Our algorithm not only assigns values to each receptive field center but also decides how many neurons in the hidden layer the network has because each such neuron contains exactly one center. The larger β is, the fewer neurons will be used in the hidden layer, which makes the training at the second stage much faster (as explained at the end of this section). However, if the number of hidden layer neurons is too small, then the mapping between the input and the output defined by the neural network loses its accuracy.

```
input:  $C = \{\mathbf{c}_{t_1}, \mathbf{c}_{t_2}, \cdots, \mathbf{c}_{t_n}\}$ and β
output: O
begin
    $O \leftarrow \varnothing$
    for each $\mathbf{c} \in C$ {
        $Temp \leftarrow$ false
        for each $\mu \in O$ {
            if ($\|\mathbf{c}-\mu\|_{\mathrm{WBC}} < \beta$) {
                $Temp \leftarrow$ true
                break
            }
        }
        if ($Temp$ == false)
            $O \leftarrow \{\mathbf{c}\} \cup O$
    }
    output O
end
```

Figure 5. The algorithm of determining the receptive field centers

Once the receptive field centers are found, we can use different heuristics to determine their widths in order to get a smooth interpolation. Park and Sandberg [25,26] show that an RBF neural network using a single global fixed value σ for all σ_j values has the capability of universal approximation. Moody and Darken [22] suggest that a good estimate of σ is the average over all distances between the center of each neuron and that of its nearest neighbor. In this paper, we use a similar heuristic to define the global width σ as

$$\sigma = \frac{1}{h}\sum_{j=1}^{h} \|\boldsymbol{\mu}_j - \boldsymbol{\mu}_j^*\|_{\mathrm{WBC}} \tag{4}$$

where *h* is the number of hidden layer neurons and $\|\boldsymbol{\mu}_j - \boldsymbol{\mu}_j^*\|_{\mathrm{WBC}}$ is the weighted bit-comparison based distance between $\boldsymbol{\mu}_j$ and its nearest neighbor $\boldsymbol{\mu}_j^*$.

After the centers and widths of the receptive fields of the RBFs in the hidden layer are determined, the remaining parameters that need to be trained are the hidden-to-output weights ($w_1, w_2, \ldots, w_h$).[7] To do so, we first select a training set composed of input coverage vectors $(\mathbf{c}_{t_1}, \mathbf{c}_{t_2}, \cdots, \mathbf{c}_{t_n})$ and the corresponding expected outputs $(r_{t_1}, r_{t_2}, \cdots, r_{t_n})$. For an input coverage vector $\mathbf{c}_{t_i}$, its actual output from the network $\hat{r}_{t_i}$ is computed as

$$\hat{r}_{t_i} = \sum_{j=1}^{h} w_j R_j(\mathbf{c}_{t_i}) \text{ and } R_j(\mathbf{x}) = \exp\left(-\frac{\|\mathbf{x} - \boldsymbol{\mu}_j\|_{\mathrm{WBC}}^2}{2\sigma_j^2}\right) \tag{5}$$

where $R_1, R_2, \ldots, R_h$ are the activation functions of the hidden layer neurons. Thus, the output of the network is:

$$\hat{\mathbf{r}} = \mathbf{A}\mathbf{w} \tag{6}$$

where $\mathbf{A} = \begin{bmatrix} R_1(\mathbf{c}_{t_1}) & R_2(\mathbf{c}_{t_1}) & \cdots & R_h(\mathbf{c}_{t_1}) \\ R_1(\mathbf{c}_{t_2}) & R_2(\mathbf{c}_{t_2}) & \cdots & R_h(\mathbf{c}_{t_2}) \\ \vdots & \vdots & \ddots & \vdots \\ R_1(\mathbf{c}_{t_n}) & R_2(\mathbf{c}_{t_n}) & \cdots & R_h(\mathbf{c}_{t_n}) \end{bmatrix}$,

$\mathbf{w} = [w_1, w_2, \cdots, w_h]^T$ and $\hat{\mathbf{r}} = [\hat{r}_{t_1}, \hat{r}_{t_2}, \cdots, \hat{r}_{t_n}]^T$

Also, let the expected output $\mathbf{r} = [r_{t_1}, r_{t_2}, \cdots, r_{t_n}]^T$ and the *prediction error* across the entire set of training data be defined as $\|\hat{\mathbf{r}} - \mathbf{r}\|_{\mathrm{E}}^2$ (the sum of squared error between $\hat{\mathbf{r}}$ and $\mathbf{r}$). To find the optimal weights $\mathbf{w}^*$, we have to compute $\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \|\hat{\mathbf{r}} - \mathbf{r}\|_{\mathrm{E}}^2 = \underset{\mathbf{w}}{\operatorname{argmin}} \|\mathbf{A}\mathbf{w} - \mathbf{r}\|_{\mathrm{E}}^2$. To achieve this objective, we use the generalized inverse (a.k.a. Moore-Penrose pseudo-inverse) of **A** [27], that is,

$$\mathbf{w}^* = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T \mathbf{r} \tag{7}$$

The complexity of computing $\mathbf{w}^*$ depends on the size of **A** which is $n \times h$, where *n* is the number of test cases in the training set and *h* is the number of hidden layer neurons. For a fixed *n*, the smaller *h* is, the smaller the complexity. Therefore, an RBF network with a smaller number of neurons in the hidden layer can be trained faster than a network with more hidden neurons.

3.3 Definition of a Weighted Bit-Comparison based Distance

From Equation (5), for a given test case t_i and its input coverage vector $\mathbf{c}_{t_i}$, the actual output $\hat{r}_{t_i}$ is a linear combination of the activation functions of all hidden layer neurons. Each R_j depends on the distance $\|\mathbf{x} - \boldsymbol{\mu}_j\|$ (referring to Equation (1)). In our case, **x** is the input coverage vector $\mathbf{c}_{t_i}$ and $\boldsymbol{\mu}_j$ is the receptive field center of the jth hidden layer neuron. So, we have $\|\mathbf{x} - \boldsymbol{\mu}_j\| = \|\mathbf{c}_{t_i} - \boldsymbol{\mu}_j\|$. From the algorithm in Figure 5, we observe that the set of receptive centers is a subset of the coverage vectors. This implies each $\boldsymbol{\mu}_j$ by itself is also the coverage vector of a certain test case. As a result, the distance $\|\mathbf{x} - \boldsymbol{\mu}_j\|$ can also be viewed as the distance between two coverage vectors.

The most commonly used distance is the Euclidean distance. However, this distance is not suitable for fault localization. For the purpose of explanation, let us use the following example. Suppose we have an RBF network trained by $\mathbf{c}_{t_1} = [0, 0, 1, 1, 0]$, $\mathbf{c}_{t_2} = [1, 0, 1, 1, 1]$ and their execution results r_{t_1} and r_{t_2}. Suppose also the trained network has two neurons in the hidden layer with $\boldsymbol{\mu}_1 = \mathbf{c}_{t_1}$ and $\boldsymbol{\mu}_2 = \mathbf{c}_{t_2}$. When we have $\mathbf{c}_{v_1} = [1, 0, 0, 0, 0]$ as the input to the trained network, the output $\hat{r}_{v_1} = w_1 R_1(\mathbf{c}_{v_1}) + w_2 R_2(\mathbf{c}_{v_1}) = w_1 \exp\left(-\frac{\|\mathbf{c}_{v_1} - \mathbf{c}_{t_1}\|^2}{2\sigma_1^2}\right) + w_2 \exp\left(-\frac{\|\mathbf{c}_{v_1} - \mathbf{c}_{t_2}\|^2}{2\sigma_2^2}\right)$, where $\sigma_1 = \sigma_2 = \sigma$. Since the first statement is covered by t_2 and v_1, but not t_1, we should have $R_1(\mathbf{c}_{v_1}) \ne R_2(\mathbf{c}_{v_1})$. This implies we should have $\|\mathbf{c}_{v_1} - \mathbf{c}_{t_1}\| \ne \|\mathbf{c}_{v_1} - \mathbf{c}_{t_2}\|$. But,

[7]In our fault localization study, the RBF network is a single-output network which produces only one output for each input coverage vector. Hence, each output layer weight (say w_j) has only one subscript, rather than two subscripts as in Equation (2), showing the connection between the output layer neuron and the corresponding hidden layer neuron (i.e., jth hidden neuron in this case).

the Euclidean distance between $\mathbf{c}_{v_1}$ and $\mathbf{c}_{t_1}$ is the same as that between $\mathbf{c}_{v_1}$ and $\mathbf{c}_{t_2}$. To overcome this problem, we use a weighted bit-comparison based distance such that

$$\| \mathbf{c}_{t_i} - \boldsymbol{\mu}_j \|_{\mathrm{WBC}} = \sqrt{1 - \cos\theta_{\mathbf{c}_{t_i}, \boldsymbol{\mu}_j}} \tag{8}$$

where $\cos\theta_{\mathbf{c}_{t_i},\boldsymbol{\mu}_j} = \dfrac{\mathbf{c}_{t_i} \bullet \boldsymbol{\mu}_j}{\|\mathbf{c}_{t_i}\|_{\mathrm{E}} \|\boldsymbol{\mu}_j\|_{\mathrm{E}}} = \dfrac{\sum_{k=1}^{m} (\mathbf{c}_{t_i})_k (\boldsymbol{\mu}_j)_k}{\sqrt{\sum_{k=1}^{m} [(\mathbf{c}_{t_i})_k]^2} \times \sqrt{\sum_{k=1}^{m} [(\boldsymbol{\mu}_j)_k]^2}}$, $(\mathbf{c}_{t_i})_k$ and $(\boldsymbol{\mu}_j)_k$ are the k^{th} element of $\mathbf{c}_{t_i}$ and $\boldsymbol{\mu}_j$, respectively. The distance measure between two binary vectors in Equation (8) is more desirable since it effectively takes into account the number of bits that are both 1 in two coverage vectors (i.e., those statements covered by both vectors). In the above example, if we replace the Euclidean distance by the weighted bit-comparison based distance, then we have two different distances as expected because $\| \mathbf{c}_{v_1} - \mathbf{c}_{t_1} \|_{\mathrm{WBC}} = 1$ and $\| \mathbf{c}_{v_1} - \mathbf{c}_{t_2} \|_{\mathrm{WBC}} = \sqrt{1/2}$.

4. An Example

Now let us demonstrate the use of the proposed method through a simple example. Suppose we have a program with ten statements s_j, $1 \le j \le 10$. Seven test cases have been executed on the program. Table 1 gives the coverage vector and the execution result of each test.

	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	r
t_1	1	1	1	1	0	1	0	0	1	1	0
t_2	1	0	0	1	1	0	1	0	0	1	0
t_3	1	1	1	0	0	1	0	0	1	1	0
t_4	1	0	1	0	0	1	1	0	1	1	0
t_5	1	1	1	0	1	0	0	0	1	0	0
t_6	1	1	1	1	0	0	0	1	1	1	1
t_7	1	0	1	1	1	1	1	1	0	1	1

Table 1. The coverage data and execution results used in the example of Section 4

We follow the six steps listed in Section 3.1. An RBF neural network with ten neurons in the input layer and one in the output layer is constructed. Using the algorithm in Figure 5 with β = 0.1, we find that each coverage vector also serves as the receptive field center of a hidden layer neuron. This implies there are seven neurons in the hidden layer. The field width σ computed using Equation (4) is 0.395. The output layer weights are trained by the data in Table 1. We have $\mathbf{w} = [w_1, w_2, w_3, w_4, w_5, w_6, w_7]^{\mathrm{T}} = [-1.326, -0.665, 0.391, -0.378, -0.308, 1.531, 1.381]^{\mathrm{T}}$. Use the coverage vectors of the virtual test cases in Part (a) of Figure 6 as the inputs to the trained network. The output with respect to each statement is shown in Part (b). These outputs are the suspiciousness of the corresponding statements.

$$\begin{bmatrix} \mathbf{c}_{v_1} \\ \mathbf{c}_{v_2} \\ \mathbf{c}_{v_3} \\ \mathbf{c}_{v_4} \\ \mathbf{c}_{v_5} \\ \mathbf{c}_{v_6} \\ \mathbf{c}_{v_7} \\ \mathbf{c}_{v_8} \\ \mathbf{c}_{v_9} \\ \mathbf{c}_{v_{10}} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Part (a): Input coverage vectors

$\hat{r}_{v_1}$	0.0384	$\hat{r}_{v_6}$	0.0179
$\hat{r}_{v_2}$	0.0481	$\hat{r}_{v_7}$	0.0157
$\hat{r}_{v_3}$	0.1246	$\hat{r}_{v_8}$	0.2900
$\hat{r}_{v_4}$	0.0768	$\hat{r}_{v_9}$	0.0066
$\hat{r}_{v_5}$	0.0173	$\hat{r}_{v_{10}}$	0.0782

Part (b): Outputs produced by the trained network which are the suspiciousness of the statements

Figure 6. Inputs and outputs/statement suspiciousness for the example of Section 4

Statements are ranked based on their suspiciousness in descending order as s_8, s_3, s_{10}, s_4, s_2, s_1, s_6, s_5, s_7, s_9. That is, s_8 is most likely to contain the bug and s_9 is least likely. We examine the statements one by one from the top until the first statement containing the bug is found.

5. Three Case Studies

5.1 Programs, Data Collection and Effectiveness

Three case studies were conducted which use the space, make and grep programs, respectively. The correct and most of the faulty versions[8] of these programs, and their test cases are downloaded from [12]. All programs are written in C. Each faulty version contains exactly one bug – the same approach is also used in many other papers [5,14,19,36,38]. We will discuss how RBFNN can be extended to programs with multiple bugs in Section 6.

For the space program, all executions were on a PC with a 2.13GHz Intel Core 2 Duo CPU and 8GB physical memory. The operating system is SunOS 5.10 (Solaris 10) and the compiler is GCC 3.4.3. For grep and make, the executions were on a Sun-Fire-280R machine with SunOS 5.10 as the operating system and GCC 3.4.4 as the compiler. Each faulty version is executed against all its corresponding available test cases. Similar to [14], multi-line statements are combined as one source code line so that they will be counted only as one executable statement. The statement coverage with respect to each test case is measured by using a revised version of χSuds [40] which could collect runtime trace correctly even if a program execution was crashed due to a "segmentation fault". The success or failure of an execution was determined by comparing the outputs of the faulty version and the correct version of a program.

Although a bug may span multiple statements which may not be contiguous or even multiple functions, the fault localization stops when the first statement containing the bug is reached. This is because our focus is to help programmers find a *starting* point to fix a bug rather than provide the complete set of code that has to be modified/deleted/added with respect to each bug. We also assume perfect bug detection, that is, a bug in a statement will be detected by a programmer if the statement is examined. If such perfect bug detection does not hold, then the number of statements that need to be examined in order to find the bug may increase. This concern applies to all the fault localization methods discussed in Section 7. In addition, we assume the cost of examining each statement for locating the bug is fixed.

The effectiveness of a fault localization method is measured by a score $\mathcal{EXAM}$ in terms of the percentage of statements that *have to be examined* until the first statement containing the bug is reached. A similar score defined as the percentage of the program that *need not* be examined to find a faulty statement is used in [14,29]. Although these two scores provide the "same" information, the $\mathcal{EXAM}$ score seems to be more direct and easier to understand. The effectiveness of different fault localization methods can be compared based on $\mathcal{EXAM}$. For a faulty version Ω, if the $\mathcal{EXAM}$ assigned to Ω by method $\mathcal{A}$ is smaller than that assigned by method $\mathcal{B}$ (that is, method $\mathcal{A}$ can guide the programmer to the fault in Ω by examining less code than method $\mathcal{B}$), then $\mathcal{A}$ is more effective than $\mathcal{B}$ for locating the bug in Ω. If there is more than one faulty version, then $\mathcal{A}$ is more effective than $\mathcal{B}$ if $\mathcal{A}$ assigns a smaller $\mathcal{EXAM}$ to more faulty versions than $\mathcal{B}$.

Results of our method are compared with those of the Tarantula method [14] for their effectiveness. The reason why the Tarantula method is chosen is because it was reported that Tarantula is more effective in fault localization than other methods such as set union, set intersection, nearest neighbor [29], and cause transitions techniques [5]. Hence, if we can show that RBFNN is more effective than Tarantula, then RBFNN is also more effective than those to which Tarantula is superior. For a fair comparison, we need to recompute the effectiveness of Tarantula using our test data and their ranking mechanism. One reason is that statistics such as fault revealing behavior and statement coverage of each test can vary under different compilers, operating systems, and hardware platforms. Also, whether the coverage measurement tool (revised χSuds versus gcc with gcov) can properly handle the "segmentation

[8]More details on the faults used in our study are explained in the subsequent sections.

fault" has an impact on the use of certain faulty versions. To ensure the validity of our re-computation, we performed cross-checks against the original reported effectiveness whenever possible. We were able to do so for the space program, but for grep and make, we did not find the effectiveness of Tarantula ready to be used.

Since the same suspiciousness may be assigned to multiple statements, this gives two different types of effectiveness: the "best" and the "worst." The "best" effectiveness assumes that the faulty statement is the first to be examined among all the statements of the same suspiciousness. For instance, supposing there are ten statements of the same suspiciousness of which one is faulty, the "best" effectiveness is achieved if the faulty statement is the first to be examined of these ten statements. Similarly, the "worst" effectiveness occurs if the faulty statement is the last to be examined of these ten statements. Hereafter, we refer to the effectiveness of RBFNN under the best and the worst cases as RBFBest and RBFWorst. Similarly, we have TBest and TWorst as the best effectiveness and the worst effectiveness of the Tarantula method.

To summarize, each study includes the following steps: instrumenting the programs, rerunning all the tests, collecting the coverage information, identifying which statements were executed by which test(s), determining whether each execution failed or succeeded, and computing the suspiciousness of each statement.

5.2 The space program

The Space program was developed at the European Space Agency. It has about 10,000 lines of C code including comments and blank lines. Jones et al reported that it has 6218 lines of executable code [14]. However, following the same convention as described in [14], if we combine multi-line statements as one source code line and count it only as one executable statement, we have 3657 executable statements. The correct and the 38 faulty versions, and a suite of 13585 test cases used in this study were downloaded from [12]. Three faulty versions were not used in our study because no test cases can reveal the faults in these versions, whereas eight faulty versions were excluded in [14].

Figure 7 gives the effectiveness of the RBFNN and Tarantula methods for the space program. The curves labeled RBFBest (in red) and RBFWorst (in blue) are for the best and the worst effectiveness of the RBFNN method, and those labeled TBest (in black) and TWorst (in green) are for the best and the worst effectiveness of the Tarantula method.[9] For a given *x* value, its corresponding *y* value is the percentage of the faulty versions whose *EXAM* score is less than or equal to *x*. The curves in this figure are drawn by connecting all the individual data points collected in our study. This is different from the curves in [14] where for each segment, a straight line is drawn between the beginning and the ending points of that segment. For 30 out of 35 faulty versions (85.71%), RBFBest can guide the programmer to the fault through examination of less than 1% of the statements. TBest only achieves such a low *EXAM* score for 22 out of 35 fault versions (62.86%). Likewise, for RBFWorst this holds for 21 of 35 faulty versions (60%), while TWorst only maintains such a low *EXAM* score for 14 out of 35 faulty versions (40%). An interesting, yet also very important, observation is that for many *x* values, even the RBFWorst is more effective than TBest. With less than 1.67% of the statements being examined, RBFWorst can guide the programmers to find bugs in 82.86% of all the faulty versions (29 faulty versions), whereas TBest can only do this for 68.57% of all the faulty versions (24 faulty versions). The same observation also applies to other *x* values. A careful comparison shows that (1) RBFBest is more effective than TBest, (2) RBFWorst is more effective than TWorst, and (3) RBFWorst is also more effective than TBest when *x* is greater than 1.15%.

[9]Since curves in Figure 7 are displayed in different colors, it is best viewed in color. This is also the case for many other figures in the paper.

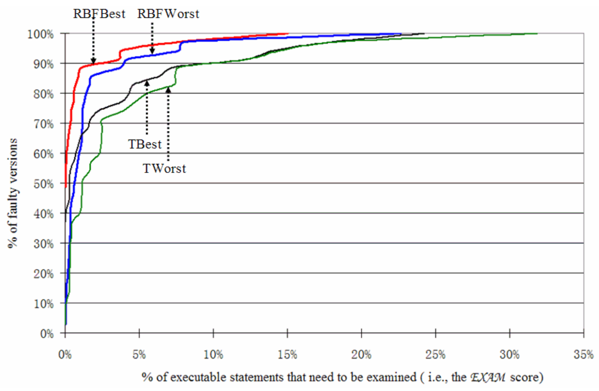


Figure 7. Effectiveness comparison between the RBFNN method and the Tarantula method for the space program

Table 2 presents the effectiveness comparison from a different perspective by showing the number of faulty versions for which RBFNN is more effective than, as effective as, and less effective than Tarantula. We observe that when RBFBest is compared with TBest, the former is more effective (i.e., examining fewer statements before the first faulty statement containing the bug is identified) than the latter for 51.43% (i.e., 18) of the 35 faulty versions, and as effective (i.e., examining the same number of statements) as the latter for 34.28% (i.e., 12) faulty versions. For only 14.29% (i.e., 5) faulty versions, RBFBest is less effective (i.e., examining more statements) than TBest. Similarly, when RBFWorst is compared with TWorst, the former is more effective than or as effective as the latter for the majority (85.71% or 30 of the 35) of the faulty versions. Even when RBFWorst is compared with TBest, still for 48.57% (17 of 35) of the faulty versions, the former is at least as effective as the latter.

	RBFBest versus TBest	RBFWorst versus TWorst	RBFWorst versus TBest
More effective	51.43% (18)	60.00% (21)	45.71% (16)
Same effectiveness	34.28% (12)	25.71% (9)	2.86% (1)
Less effective	14.29% (5)	14.29% (5)	51.43% (18)

Table 2. Pairwise comparison between RBFNN and Tarantula for the space program

A third way to compare the effectiveness of RBFNN and Tarantula is based on the total number of statements that have to be examined by each method for all 35 faulty versions of the space program. Referring to Table 3 the total number of statements examined by RBF is between 1337 (best case) and 2417 (worst case), whereas the total number of statements examined using Tarantula is between 3876 (best) and 5094 (worst). A closer examination shows that RBFBest examines 65.51% fewer statements than TBest, and RBFWorst examines 52.55% fewer statements than TWorst. Moreover, RBFWorst examines 37.64% fewer statements than TBest.[10]

	RBFBest	RBFWorst	TBest	TWorst
Number of statements	1337	2417	3876	5094

Table 3. Total number of statements examined for all 35 faulty versions of space

5.3 The grep program

The grep program searches a file for a pattern. The source code of version 2.2 was downloaded from [12]. It has 12653 lines of C code including comments, blank lines, etc. and 3306 executable statements. Also downloaded were a suite of 470 test cases and 18 bugs. Compared with the study in [19] where none of these bugs

[10]Note that the set of statements examined by RBFBest is not necessarily a subset of the statements examined by TBest. The same applies to RBFWorst and TWorst. This is true for all three case studies.

could be detected by any test case in the suite, there are only 14 that could not be detected in our environment. Two additional bugs injected by Liu et al. [19] were also used. We followed a similar approach to inject 13 more bugs. Altogether, there are 19 faulty versions each of which contains exactly one bug. Manually injected faults are designed to mimic realistic bugs, as described in [19]. For example, errors such as "subclause-missing" or "off-by-one" tend to manifest when programmers are unclear about the corner condition. A list of all the bugs used in our study is available upon request.

Figure 8 shows the effectiveness of RBFNN and Tarantula for the grep program where all the legends, the horizontal axis, and the vertical axis have the same meaning as in Figure 7. It is clear that RBFBest is more effective than TBest, and RBFWorst is more effective than TWorst. Also, RBFWorst is more effective than TBest with respect to many $\mathcal{EXAM}$ scores.

From Table 4, we observe that RBFBest is more effective than or as effective as TBest for 89.47% (i.e., 17) of the 19 faulty versions of the grep program. For only 10.53% (i.e., 2) faulty versions, RBFBest is less effective than TBest. The same applies to the comparison between RBFWorst and TWorst. Even when RBFWorst is compared with TBest, still for 52.63% (10 of 19) of the faulty versions, the former is at least as effective as the latter.

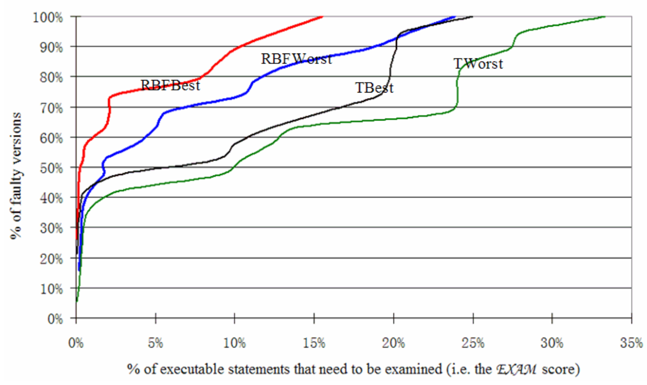


Figure 8. Effectiveness comparison between the RBFNN method and the Tarantula method for the grep program

	RBFBest versus TBest	RBFWorst versus TWorst	RBFWorst versus TBest
More effective	73.68% (14)	78.94% (15)	47.37% (9)
Same effectiveness	15.79% (3)	10.53% (2)	5.26% (1)
Less effective	10.53% (2)	10.53% (2)	47.37% (9)

Table 4. Pairwise comparison between RBFNN and Tarantula for the grep program

Table 5 shows that not only RBFBest examines 64.18% fewer statements than TBest and RBFWorst examines 49.26% fewer statements than TWorst but also RBFWorst examines 31.57% fewer statements than TBest.

	RBFBest	RBFWorst	TBest	TWorst
Number of statements	2075	3964	5793	7812

Table 5. Total number of statements examined for all 19 faulty versions of grep

5.4 The make program

The make program is a software utility that manages the building of executables and other products from source code. Version 3.76.1 of make was downloaded from [12]. The program has a total of 20014 lines of C code, with 5318 executable statements after excluding comments, blank lines, etc. 793 test cases were provided, along with 19 faulty versions of the program. Of these, 15 faulty versions were excluded as they contained bugs which could not be detected by any of the downloaded test cases in our environment. Using a similar fault injection approach to that described in [19], we generated an additional 27 bugs for a total of 31 usable faulty versions. As mentioned in Section 5.3, the manually injected faults are designed to mimic realistic program errors.

Figure 9 shows the effectiveness of RBFNN and Tarantula for the make program where all the legends, the horizontal axis, and the vertical axis have the same meaning as in Figure 7. We observe that for any x value RBFBest is more effective than TBest, and RBFWorst is more effective than TWorst. Also, RBFWorst is more effective than TBest with respect to many $\mathcal{EXAM}$ scores.

From Table 6, we observe that RBFBest is more effective than or as effective as TBest for 90.32% (i.e., 28) of the 31 faulty versions of the grep program. For only 9.68% (i.e., 3) faulty versions, RBFBest is less effective than TBest. The same applies to the comparison between RBFWorst and TWorst. Even when RBFWorst is compared with TBest, still for 51.61% (16 of 31) of the faulty versions, the former is at least as effective as the latter.

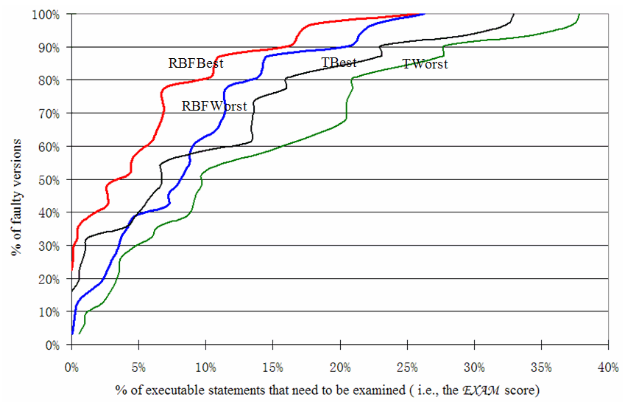


Figure 9. Effectiveness comparison between the RBFNN method and the Tarantula method for the make program

	RBFBest versus TBest	RBFWorst versus TWorst	RBFWorst versus TBest
More effective	74.19% (23)	90.32% (28)	51.61% (16)
Same effectiveness	16.13% (5)	0	0
Less effective	9.68% (3)	9.68% (3)	48.39% (15)

Table 6. Pairwise comparison between RBFNN and Tarantula for the make program

Table 7 shows that not only RBFBest examines 45.60% fewer statements than TBest and RBFWorst examines 37.83% fewer statements than TWorst but also RBFWorst examines 13.62% fewer statements than TBest.

	RBFBest	RBFWorst	TBest	TWorst
Number of statements	9188	14590	16890	23468

Table 7. Total number of statements examined for all 31 faulty versions of make

5.5 Overall Observation

In general, we would expect that the effectiveness of RBFNN lies between RBFBest and RBFWorst. Similarly, the effectiveness of Tarantula lies between TBest and TWorst. It is impractical to assume that the best effectiveness could be achieved every time. In fact, our intuition suggests that when the size of the program being debugged increases, it is more likely for RBFNN and Tarantula to group more statements together with the same suspiciousness. This also makes it even less likely to have the faulty statement to be the first one being examined among all the statements with the same suspiciousness. Based on the comparisons using the experimental data collected from our studies on the space, grep and make programs, we observe that not only RBFBest is more effective than TBest and RBFWorst more effective than TWorst but also

RBFWorst more effective than TBest in many cases. This clearly indicates that RBFNN is more effective in fault localization (i.e., examining less code before the first faulty statement is located) than Tarantula in our studies.

6. Programs with multiple bugs

Although each faulty version in the three case studies reported in Section 5 contains exactly one fault, we can also apply RBFNN to programs with multiple bugs. This is done in two steps. The first step is to conduct an appropriate clustering on failed executions or failed tests. Different techniques have been proposed for this purpose [13,20,28,44]. For example, we can group failed test cases into *fault-focusing* clusters such that those in the same cluster are related to the same fault [13]. The second step is to combine failed tests in each cluster with the successful tests for debugging a single fault. More specifically, we can follow a similar approach as described in [13] to generate a specialized test suite for each fault. The difference is that instead of using Tarantula's ranking mechanism, we can use the method discussed in Section 3 to rank the statements. As a result, our RBFNN method can be extended for debugging programs with multiple bugs.

We conducted a case study using GCC (the GNU Compiler Collection) to prove the *concept* discussed above. There are multiple releases of GCC posted at the GNU website (http://gcc.gnu.org). In our study, GCC 3.4.1 was used and the 29 C files in its sub-directory gcc/cp were instrumented using a revised version of χSuds [40]. These files contain 95218 lines. After combining each individual multi-line statement into one executable statement and deleting all the non-executable statements (comments, blank lines, function and variable declarations, etc.), there are 30531 executable statements in C. We do not have a separate version of GCC each of which contains exactly one bug. Instead, we have only one GCC program with multiple bugs. There are 9489 successful test cases for this version.

Defect data retrieved from the GCC Bugzilla database are used to determine the exact location of each bug. Information of additional test case(s) created for each bug is also available from bug reports to help us determine the failed tests for each bug. We use such information to accomplish the *fault-focusing* clustering in step 1 to group failed test cases into clusters that target different bugs [13]. A more robust clustering technique will be developed in our future study; however, that is out of the scope of this paper. Each bug-focusing cluster is combined with the successful tests for locating a single bug. Five bugs were used in our study: $\mathcal{B}_1$ (ID 16637), $\mathcal{B}_2$ (ID 16889), $\mathcal{B}_3$ (ID 16929), $\mathcal{B}_4$ (ID 16965) and $\mathcal{B}_5$ (ID 18140). Each of bugs $\mathcal{B}_1$, $\mathcal{B}_2$, $\mathcal{B}_3$ and $\mathcal{B}_4$ has one failed test and 9489 successful tests; $\mathcal{B}_5$ has two failed tests and 9489 successful tests. The execution environment for GCC is the same as that for the space program (see Section 5.1).

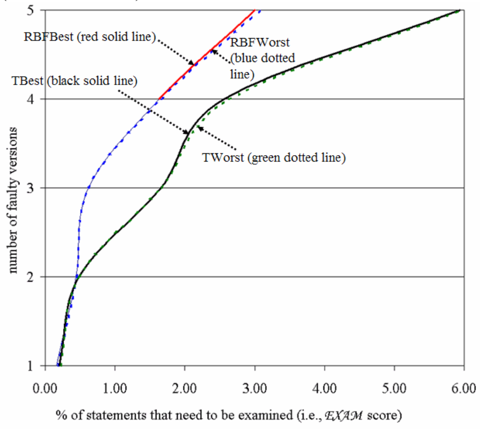


Figure 10. Effectiveness comparison between the RBFNN method and the Tarantula method for GCC

Figure 10 shows the effectiveness of RBFNN and Tarantula for GCC where all the legends, and the horizontal axis have the same meaning as in Figure 7, but the vertical axis is the number instead of the percentage of faulty versions. Table 8 presents the pairwise effectiveness comparison and Table 9 lists total number of statements examined by RBFNN and Tarantula for the five bugs of GCC. Our data suggest that for these bugs, RBFNN is more effective than Tarantula. Once again the purpose is only to demonstrate that RBFNN can be applied to programs with multiple bugs as long as we have a way to do the fault-focusing clustering. More meaningful case studies on programs with multiple bugs are currently ongoing.

	RBFBest versus TBest	RBFWorst versus TWorst	RBFWorst versus TBest
More effective	4	4	4
Same effectiveness	0	0	0
Less effective	1	1	1

Table 8. Pairwise comparison between RBFNN and Tarantula for GCC

	RBFBest	RBFWorst	TBest	TWorst
Number of statements	1789	1826	3315	3352

Table 9. Total number of statements examined for the 5 faulty versions of GCC

7. Discussion

In this section, we discuss some important aspects related to our RBF-based fault localization method.

7.1 RBFNN and Two Statistical Debugging Methods

Liblit et al. [17] presented a statistical debugging algorithm for deployed programs (referred to hereafter as Liblit05). Liu et al. proposed the SOBER model [19] to rank suspicious predicates. The major difference between our RBFNN, SOBER, and Liblit05 is that we use a modified RBF neural network to rank suspicious statements, whereas the last two rank suspicious predicates for fault localization. For them, the corresponding statements of the top *k* predicates are taken as the initial set to be examined for locating the bug. As suggested by Jones and Harrold in [14], Liblit05 provides no way to quantify the ranking for all statements. An ordering of the predicates is defined, but the approach does not expand on how to order statements related to any bug that lies outside a predicate. For SOBER, if the bug is not in the initial set of statements, additional statements have to be included by performing a breadth-first search on the corresponding program dependence graph. However, this search is not required using RBFNN as all the statements of the program are ranked based on their suspiciousness (Section 3.1). Once the network is trained, computing the suspiciousness of each statement and ranking them in a descending order based on the suspiciousness is very efficient.

Below, we present a quantitative comparison on the effectiveness among RBFNN, SOBER and Liblit05. We conducted an additional case study using the Siemens suite. Of the 132 faulty versions, three were not used because (1) none of the test cases downloaded from http://www-static.cc.gatech.edu/aristotle/Tools/subjects can reveal the fault in version 9 of "schedule2," and (2) the faults in versions 4 and 6 of "print_tokens" are in the header files instead of in the C files. Note that in [14] (a study of applying Tarantula to the Siemens suite), ten faulty versions (including the three mentioned above) were excluded for various reasons. Based on the effectiveness calculated using our data, we observe that RBFBest, RBFWorst, TBest and TWorst can identify 75, 63, 67, and 58 bugs, respectively, by examining no more than 10% of the code. Due to the time constraint, we did not recalculate the effectiveness of SOBER and Liblit05 using our data. Instead, we used the data reported in [19] which says of the 130 bugs in the Siemens suite, SOBER can help programmers locate 68 bugs by examining no more than 10% of the code and Liblit05 locates 52 bugs. Note that two bugs (the 32nd version of "replace" and the 10th version of "schedule2") were excluded. From these data, we make the following observations

- Both RBFBest and RBFWorst are better than Liblit05
- RBFBest is better than TBest and RBFWorst is better than TWorst
- RBFBest is better than SOBER, but RBFWorst is worse than SOBER
- Both TBest and TWorst are worse than SOBER

where "better" implies "more bugs can be located," and "worse" implies "fewer bugs can be located." There is no information reported in [19] on how many bugs can be located using SOBER and Liblit05 by examining more than 10% of code. Similarly, we cannot find existing data on effectiveness using SOBER and Liblit05 for the "space," "grep," and "make" programs. This makes a quantitative comparison on effectiveness of these fault localization methods more difficult.

7.2 RBFNN and Other Machine Learning Algorithms

The RBF approach presented in this paper has important general features in common with a variety of widely used machine learning algorithms. In the following, we briefly review a few representative machine learning algorithms which are closely related to our approach.

Our methodology is similar to a support vector machine (SVM) methodology in that an input vector is also mapped into a feature space (in this case the features are the neurons at the hidden layer of the modified RBF) and then a linear model is used to compute a weighted sum of features [10]. An SVM is generally used for classification purposes. If the weighted sum exceeds a threshold value, then the input is classified as exceeding the threshold value; otherwise the input is classified as not exceeding the threshold value. Also, SVMs typically use Vapnik's epsilon-insensitive loss function for parameter estimation. However, we used a negative log-likelihood loss function with respect to a conditional Gaussian regression model for parameter estimation. Besides, our focus is not on "classification" of inputs, but on the value of an SVM linear discriminant function, $\hat{r}_{v_i}$, which is the measure of suspiciousness of the ith statement. In our subsequent studies, we may choose to use the Vapnik loss function to explore the possible improvement of generalization performance of our methodology.

Decision tree methods partition the input vector space into a set of hyperrectangles and a linear model is used to compute a weighted sum of features. The novel aspect of decision tree methods is that algorithms have been developed for splitting rectangles and creating decision trees with different types of methodologies. Our modified RBF may be interpreted as partitioning the input space into a set of hyperellipsoids and again a linear model is used to compute a weighted sum of features [10]. There is no intrinsic advantage of using a decision space of hyperrectangular features versus a decision space of hyperellipsoidal features. The advantage of one decision space over another will ultimately be determined by characteristics of the statistical data. In general, there is no single machine learning algorithm which provides the optimal generalization performance for all data sets. We intend to conduct future theoretical and/or empirical investigations to better understand which type of decision space might be ultimately preferable for our problem, but we have chosen to use hyperellipsoidal features in this initial investigation to illustrate the essential novel ideas (discussed in Section 1) of our approach.

Generalized logistic regression methods also involve having an input vector mapped into a feature space and then a weighted sum of features is computed as well. For these methods, the weighted sum of features is mapped via a logistic sigmoid transformation into the probability of a particular classification [10]. In our approach, such mapping is done by using linear regression so that the output is the conditional expectation of the suspiciousness given the corresponding coverage vector. We intend to use logistic regression in our subsequent investigations but we have chosen to explore the more straightforward generalized linear regression model methodology initially.

In summary, since this is the first feasibility study of our proposed fault localization method, instead of using complicated algorithms which may distract readers from the nature of the problem (namely, effective fault localization), we feel it is better to use a more straightforward machine learning algorithm (such as the modified RBF presented here) to demonstrate both the novelty and validity of our approach.

7.3 Limitation and Threats to Validity

An important limitation of our method is that the number of hidden neurons, receptive field centers, and weights are not simultaneously estimated. A potential problem with the sequential estimation method used here is that the performance of the network is suboptimal and could be improved further. We did not implement a simultaneous estimation methodology for two reasons. First, a simultaneous estimation implies the existence of a multi-modal objective function (possibly with multiple local minima and saddle points) making the evaluation of learning solutions substantially more complex. Second, the method implemented here can be interpreted as a generalized linear regression model with highly nonlinear preprocessing transformations which is a reasonable baseline approach for a feasibility study.

Another limitation is that the mapping from the activation pattern over radial basis function hidden neurons to the output neuron activation level is modeled as a linear regression rather than a logistic regression. Or, equivalently, selecting the activation function of the output neuron to be a logistic sigmoidal function rather than a linear function would allow the output of the neural network to be interpreted as the probability of a failed execution result. Although improved generalization performance would be expected by using this slightly more complicated logistic regression modeling, it was decided in this study to try a relatively simple approach intended to illustrate the essential novel ideas before exploring these more sophisticated methods.

We would also like to emphasize that like any other fault localization techniques, the effectiveness of RBFNN varies for different programs, bugs, and test cases. The coverage measurement tools (such as whether the runtime trace can be correctly collected even if a program execution is crashed due to a "segmentation fault") and environments (including compilers, operation systems, hardware platforms, etc.) also have an impact.

As for how real-world developers can benefit from our method, it is still yet to be answered. In our subsequent studies, applying RBFNN to an industry environment is one of the top priorities.

8. Related Studies

Different techniques have been proposed for fault localization such as (1) the traditional approaches (e.g., inserting *print* statements or analyzing the memory dump), (2) slicing-based (e.g., static slicing-based [21,34], dynamic slicing-based [1,15], and execution slicing-based [2,35,37]), (3) state-based (e.g., delta debugging [41,42], cause transition [5], predicate switching [43]), spectrum-based (e.g., those using executable statement hit spectra, branch hit spectra, etc.), statistical model-based (e.g., Liblit et al. [17]), SOBER [19] and Crosstab-based [39]).

Due to the space limit, in this paper we focus on Tarantula [14] and a few other techniques (set union, set intersection, nearest neighbor and cause transitions) to which Tarantula has been reported to be superior. Similar to RBFNN, Tarantula also uses the coverage and execution results to compute the suspiciousness of each statement as X/(X+Y) where X = (number of failed tests that execute the statement)/(total number of failed tests) and Y = (number of successful tests that execute the statement)/(total number of successful tests). A recent study [13] applies Tarantula to programs with multiple bugs. Refer to Section 6 for more details.

The set union technique [29] computes the set difference between the program spectra of a failed test and the union spectra of a set of successful tests. The set intersection technique [29] excludes the code that is executed by all the successful tests but not by the failed test. The nearest neighbor debugging approach [29] compares a failed test with another successful test which is most similar to the failed one in terms of the "distance" between them. The cause transition debugging approach [5], based on delta debugging [41,42], examines program state to determine the locations where the cause of failure changes from one variable to another.

9. Conclusion and Future Work

An RBF neural network-based fault location method is presented. The training set for the network consists of the coverage information for each test case paired with its execution result (either success or failure). Once the trained RBF network has been prepared, it is given as input a set of virtual test cases, each of which covers a single statement. The suspiciousness of a statement is considered to be the output of the network for the virtual test case covering that statement. Statements with a higher suspiciousness should be examined first as they are more likely to contain program bugs. The effectiveness of our method is measured using the *EXAM* score in terms of the percentage of statements that *have to be examined* until the first statement containing the bug is reached.

Experimental data based on space, grep, make (programs with each faulty version containing exactly one bug) and gcc (a program with multiple bugs) suggest that RBFNN is more effective in fault localization than Tarantula, which also uses the coverage and execution results to compute the suspiciousness of each statement. However, we understand that it is not possible to generalize this conclusion based on the results from a small number of case studies. The effectiveness of a fault localization technique can be influenced by a variety of factors, including the program structure, the nature of the faults, and the composition of the test suite. Studies that target a wider range of application domains are currently in progress to further validate the general effectiveness of our fault localization technique. We will also extend studies using other machine learning algorithms (e.g., support vector machines, decision trees, logistic regression, etc.) and observe the potential variation of the performance. Simulation resampling methods such as parametric bootstrap and non-parametric bootstrap methods are considered as well to characterize the generalization performance of RBFNN in a more explicit manner.

Also in development is an improved method to cluster failed tests and executions according to their underlying fault. In a program with multiple bugs, a cluster of failed tests along with the complete set of successful tests can then be used to train an RBF network to locate the fault associated with that cluster.

References:

1. H. Agrawal, R. A. DeMillo, and E. H. Spafford, "Debugging with dynamic slicing and backtracking," *Software: Practice & Experience*, 23(6):589-616, June, 1996.
2. H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault localization using execution slices and dataflow tests," in *Proc. of ISSRE*, 1995.
3. G. Boetticher and D. Eichmann, "A neural network paradigm for characterizing reusable software," in *Proc. of the 1st Australian Conf. on Software Metrics*, pp. 41-54, Sydney, Australia, 1993.
4. S. R. Chu, R. Shoureshi, and M. Tenorio, "Neural networks for system identification," *IEEE Control Systems Magazine*, 10(3):31-35, 1990.
5. H. Cleve and A. Zeller, "Locating causes of program failures," in *Proc. of ICSE*, 2005.
6. J. Dang, Y. Wang, and S. Zhao, "Face recognition based on radial basis function neural networks using subtractive clustering algorithm," in *Proc. of the Sixth World Congress on Intelligent Control and Automation*, pp. 10294-10297, 2006.
7. K. Fukushima, "A neural network for visual pattern recognition," *Computer*, 21(3):65-75, March 1998.
8. M. T. Hagan, H. B. Demuth, and M. Beale, "Neural network design," PWS Publishing, 1995.
9. M. H. Hassoun, "Fundamentals of Artificial Neural Networks," The MIT Press, 1995.
10. T. Hastie, R. Tibshirani, and J. Friedman, "The Elements of Statistical Learning: Data Mining, Inference, and Prediction," Springer, 2001.
11. S. Haykin, "Neural Networks: A Comprehensive Foundation (2nd Edition)," Prentice Hall, 1999.
12. http://sir.unl.edu/portal/index.html
13. J. A. Jones, J. Bowring, and M. J. Harrold, "Debugging in parallel," in *Proc. of ISSTA*, 2007.
14. J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," in *Proc. of ASE*, 2005.
15. B. Korel and J. Laski, "Dynamic program slicing," *Information Processing Letters*, 29(3):155-163, October 1988.
16. C. C. Lee, P. C. Chung, J. R. Tsai, and C. I. Chang, "Robust radial basis function neural networks," *IEEE Trans. on Systems, Man, and Cybernetics: Part B Cybernetics*, 29(6):674-685, December 1999.
17. B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proc. of PLDI*, 2005.
18. G. F. Lin and L. H. Chen, "Time series forecasting by combining the radial basis function network and the self-organizing map," *Hydrological Processes*, 19(10):1925-1937, June 2005.
19. C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: a hypothesis testing-based approach," *IEEE TSE*, 32(10):831-848, 2006.
20. C. Liu and J. Han, "Failure proximity: a fault localization-based approach," in *Proc. of FSE*, 2006.
21. J. R. Lyle and M. Weiser, "Automatic program bug location by program slicing," in *Proc. of the 2nd Intl. Conf. on Computer and Applications*, pp. 877-883, 1987.
22. J. Moody and C. J. Darken, "Learning with localized receptive fields," in *Proc. of Connectionist Models Summer School*, pp. 133-142, 1988.
23. K. S. Narendra and S. Mukhopadhyay, "Intelligent control using neural networks," *IEEE Control System Magazine*, 12(2):11-18, 1992.
24. D. E. Neumann, "An enhanced neural network technique for software risk analysis," *IEEE TSE*, 28(9):904-912, 2002.
25. J. Park and I. W. Sandberg, "Universal approximation using radial-basis-function Networks," *Neural Computation*, 3(2), 1991.
26. J. Park and I. W. Sandberg, "approximation and radial-basis-function networks," *Neural Computation*, 5(2):305-316, 1993.
27. R. Penrose, "A generalized inverse for matrices," *Proceedings of the Cambridge Philosophical Society*, 51, pp. 406-413, 1955.
28. A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated support for classifying software failure reports," in *Proc. of ICSE*, 2003.
29. M. Renieres and S. P. Reiss, "Fault localization with nearest neighbor queries," in *Proc. of ASE*, 2003.
30. Y. S. Su and C. Y. Huang, "Neural-network-based approaches for software reliability estimation using dynamic weighted combinational models," *Journal of Systems and Software*, 80(4):606-615, 2007.
31. N. Tadayon, "Neural network approach for software cost estimation," in *Proc. of Intl. Conf. on Information Technology: Coding and Computing*, pp. 815- 818, Las Vegas, Nevada, April 2005.
32. C. Wan and P.B. Harrington, "Self-configuring radial basis function neural networks for chemical pattern recognition," *Journal of Chemical Information and Modeling*, 39(6):1049-1056, November 1999.
33. P. D. Wasserman, "Advanced Methods in Neural Computing," Van Nostrand Reinhold, 1993.
34. M. Weiser, "Programmers use slices when debugging," *Communications of the ACM*, 25(7):446-452, July 1982.
35. W. E. Wong and Y. Qi, "Effective program debugging based on execution slices and inter-block data dependency," *Journal of Systems and Software*, 79(7):891-903, July 2006.
36. W. E. Wong, Y. Qi, and L. Zhao, "Effective fault localization using code coverage," in *Proc. of COMPSAC*, 2007.
37. W. E. Wong, T. Sugeta, Y. Qi, and J. C. Maldonado, "Smart debugging software architectural design in SDL," *Journal of Systems and Software*, 76(1):15-28, April 2005.
38. W. E. Wong, L. Zhao, and Y. Qi, "Fault localization using BP neural networks," in *Proc. of SEKE*, 2007.
39. W. E. Wong, T. Wei, Y. Qi, and L. Zhao, "A Crosstab-based Statistical Method for Effective Fault Localization," in *Proc. of ICST,* 2008.
40. χSuds User's Manual, Telcordia Technologies, 1998.
41. A. Zeller, "Isolating cause-effect chains from computer programs," in *Proc. of ACM Symp. on Foundations of Soft. Engg.*, pp. 1-10, 2002.
42. A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE TSE*, 28(2):183-200, 2002.
43. X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *Proc. of ICSE*, 2006.
44. A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken, "Statistical debugging: simultaneous identification of multiple bugs," in *Proc. of Intl. Conf. on Machine Learning*, pp. 1105-1112, 2006.