MIS Review Vol. 15, No. 2, March (2010), pp. 75-90

Non Homogenous Poisson Process Model for Optimal Software Testing Using Fault Tolerance

Praveen Ranjan Srivastava, Chetan Mittal, Ajay Rungta, Vaibhav Malhotra, G Raghurama
Birla Institute of Technology and Science, Pilani, Rajasthan, India

ABSTRACT: *In software industry, it is important to prioritize the different modules of a software so that important modules are tested ahead of the lesser important ones. This approach is desirable because it is not possible to test each module regressively due to time and cost constraints. This paper proposes a way to prioritize several modules of a software product and calculates optimal time and cost for testing based on non homogenous poisson process. Sometimes it is more profitable for an organization to release software, even if it is not completely tested because of limited time and resources. This paper also tries to figure out whether the software could be released or not, after testing within a given time and cost.*

KEYWORDS: *Non Homogenous Poisson Process, Optimal Test Policy, Software Life Cycle Length, Testing Time, Module Test Prioritization, Fault Tolerance.*

1. Introduction

The essence of software testing is to find out any faults that might exist before releasing the product in the market. For this purpose, software product is tested carefully. The very primitive method of testing software is regression testing [15]. It is the process of testing software to make sure that old program still works with the new changes. Regression testing is any type of software testing which seeks to uncover software regressions. Such regressions occur whenever software functionality that was previously working correctly, stops working as intended. Typically regressions occur as an unintended consequence of program changes. Common methods of regression testing include re-running previously run tests and checking whether previously fixed faults have re-emerged [15]. But it is not feasible to perform regression testing on the software always, as it can be very expensive. In fact a large portion of the software maintenance budget can be consumed by regression testing [1]. That's why, a tester should find out what are the modules with greater importance so that they can be tested first and given more effort. It is impractical to test the software unless all the bugs are removed. The tester should also be aware of the optimal testing time and cost required to test the modules. When it is not possible to remove all the bugs with limited resources, then we have to accept limited faults in the software. For this reason, this paper attempts to provide an optimal boundary values for time and cost considering the actual percentage of

faults obtained in testing. A project manager should know to where it should stop testing and go for release or rejection.

A lot of work has been done in the area of optimal software time calculation by McDaid and Wilson who gave three plans to settle on the optimal time [2]. Musa and Ackerman used the concept of reliability to make the decision [3]. Ehrlich, Prasanna, Stampfel and Wu tried to find out the cost of a stop test decision [4]. But one of the most suitable models for determining optimal cost and time is proposed by Goel and Okumoto [7]. They have suggested for a non homogenous poisson process based model to determine the optimal cost and time for software [5] [6]. Praveen et al. have proposed a cumulative priority based elucidation to find out optimal software testing period [8].

Till now, the focus has been on the testing time and cost only. Previous works assume that there are errors in the software if the actual testing time is greater than the estimated time. But this assumption is incorrect when actual testing time is higher due to bad testing practices or some other reasons but not due to faults. So we can say, large number of errors means more testing time but more testing time does not mean large number of errors. This paper is also considering faults along with time and cost. The aim of this paper is to classify different modules into 5 precedence categories and to find out whether the software is ready to be released in the market after testing it for the given time within specified cost.

The next section briefly explains the background and the related work. Section 3 provides the module prioritization schema based on various factors and our approach to test the software to determine if it is tested enough. Section 4 brings an example where this approach is applied. Last section attempts to draw a conclusion.

2. Background and related work

2.1 Non homogeneous poisson process

A Poisson process is one of the most significant random processes in the probability theory. It is widely used to model random points in time and space. The examples include the times of radioactive emissions, the arrival times of customers at a service center and the positions of flaws in a piece of material. Several important probability distributions arise naturally from the Poisson process. The Poisson process is a collection of random variables where N(t) is the number of events that occurred up to time t (starting from time 0) [8]. The number of events between time a and time b is given as N(b) − N(a) and has a Poisson distribution. A Non-Homogeneous process is dependent on rate parameter λ(t) where the rate parameter of the process is a function of time e.g. the arrival rate of vehicles in

a traffic light signal. Here, we need to understand that software bugs also follow the non homogenous poisson process as the arrival of bugs in software development life cycle is random. In the following section, we can see how Goel and Okumoto used non homogenous poisson process to estimate total cost and time.

2.2 Related work by Goel and Okumoto [5][6][7]

Faults present in the system causes software failure at random times. Let N(t) (where t > 0) be the cumulative number of failures at time t (can be either CPU time or calendar time). According to Goel and Okumoto, software failure process m(t) i.e. expected number of faults detected by time t can be shown as (1):

$$m(t) = a(1-e^{-bt}) \quad (1)$$

Here, $m(\infty) = a$ where "a" represents the expected number of software failures to be eventually encountered and "b" is the detection rate for an individual fault.

According to Goel and Okumoto operational performance of a system largely depends upon testing time. Longer testing phase leads to enhanced performance. Also, cost of fixing a fault during operation is generally much more than during testing. However, the time spent in testing delays the product release, which leads to additional costs. The objective is to determine optimal release time to minimize cost by reducing testing time. Goel and Okumoto have designed the parameters c1, c2, c3, t and T which are as follows:

- c1 = cost of fixing a fault during testing
- c2 = cost of fixing a fault during operation (c2>c1)
- c3 = cost of testing per unit time
- t = software life cycle length
- T = software release time (same as testing time)

Since m(t) represents the expected number of faults during (0,t) the expected costs of fixing faults during the testing and operational phases are c1m(T) and c2(m(t)-m(T)) respectively. Further, the testing cost during a time period T is c3(T). If there is a cost associated with delay in meeting a delivery plan, such a cost could be included in c3. Combining the above costs, the total expected cost is given by (2).

$$C(T) = c1m(T) + c2\,(m(t) - m(T)) + c3(T) \quad (2)$$

This policy minimizes the average cost and depends on the ratio of a*b and Cr = c3 / (c2-c1). (3)

Two cases arise, a*b > Cr and a*b <= Cr

(i) If a*b > Cr, the optimal policy is to take

$T^* = \min (T_0, t)$ (4)

Where $T_0 = 1/b \ln(a*b / Cr)$

(ii) If $a*b <= Cr$, then $T = 0$.

If the cost of testing or the cost of delay in release is very high, this work tend to "No Testing" at all i.e. $T^* = 0$. On the other hand, if the cost of fixing a fault after release is very high as compared to the usefulness of the system, one would prefer not using the system i.e. $T^* = t$.

2.3 Related work by Praveen et al. [8][12][13]

Praveen et al proposes prioritizing the software modules into five categories namely very high, high, medium, low and very low. These categories decide the rank order of the modules to be tested in the descending order i.e. very high category modules will be tested before high and so on. Then it calculates optimal cost and time from the Goel and Okumoto work. To find out maximum allowable cost and time, stringency concept is used here. Stringency is the maximum allowable deviation from the optimum which is decided by the organization.

They advise to start testing the software to calculate the actual time and actual cost for each priority category. The deviation from optimal testing time and optimal cost can be calculated from (5) and (6). [8]

$\alpha = (Ta - T^*)/T^*$ (5)

Where α = deviation from optimal time

Ta = actual testing time

T^* = optimal testing time calculated from (4)

And $\beta = (Ca - Co)/Co$ (6)

Where β = deviation from optimal cost

Ca = actual testing cost

Co = optimal testing cost calculated from (2)

Limiting factor δ is given by (7)

$\delta = \alpha + \beta$ (7)

Afterwards it cumulatively calculates the limiting factor δ to determine whether further software testing is required.

3. Proposed approach and work

3.1 Module test prioritization schema

It is crucial to introduce a schema which ensures that the component prioritization is uniform and effective [13]. When all the modules are complete, there is a need to prioritize them for testing. Sometimes we don't have enough resources to test all the modules exhaustively. In that case we prioritize them so that modules with high priority can be tested earlier than the low priority modules. We have used the following parameters for module prioritization:

Person hour -- This is the amount of work carried out by an employee. Organization can keep track of total person hours for a module. Module priority will increase as person hours increases.

Decision density -- It can be calculated by dividing total Cyclomatic Complexity (CC) by logical lines of code [9]. Logical line of code is actual source code excluding empty lines and comment lines. Total Cyclomatic Complexity (TCC) for a module is computed by (8):

$$TCC = Sum(CC) - Count(CC) + 1 \quad (8)$$

In other words, CC is summation of all procedures. Count (CC) equals the number of procedures. The importance of TCC can be seen from this example. Suppose, there are 4 'if' decisions in a procedure, so its CC will be 5 (number of decisions +1) and its TCC from eq. (8), will be 5 (as 5 -1 +1). Now, say this procedure is split into 2 different procedures having 2 'if' conditions each i.e. a CC of 3 each. In this case also the TCC comes out to be 5 (as 6 -2 +1). So, the TCC is unaffected for the same piece of code regardless of the code split.

Weight priority -- This includes ranking given by developers, managers and customer based on the requirements and the ranking based on the risk factors. [10] Ranks are given within the range 1 to 10 for both the categories i.e. requirements and risks. There are weight factors associated with both of them in such a way that sum of these weights is 1. For example the weight factor for requirement is 0.6 and risk is 0.4. Now, say a module has requirements rank as 7 and risk rank as 8, then its total rank would be 0.6(7) + 0.4(8) = 7.4. The higher this rank is, the higher the importance would be given to the module.

Code reusability -- If an earlier source code is used in the current work with little or no modifications then we call it code reusability. This lessens the requirements of testing the code again as it has already been tested earlier.

Coupling -- It is the measure of connectedness of one module to other [11]. It is given as (9):

$$C = 1 - 1 / (d_i + 2 \cdot c_i + d_o + 2 \cdot c_o + g_d + 2 \cdot g_c + w + r) \quad (9)$$

Where C = Coupling

d_i = number of input data parameters

c_i = number of input control parameters

d_o = number of output data parameters

c_o = number of output control parameters

g_d = number of global variables used as data

g_c = number of global variables used as control

w = number of modules called (fan-out)

r = number of modules calling the module under consideration (fan-in)

Our suggested formula for calculating Module Priority is given in (10) below.

$$MP = w1 * rel.PH + w2 * rel.DD + w3 * rel.WT - w4 * rel.\%CR + w5 * rel.MC + 1 \quad (10)$$

Where MP = module priority

rel. PH = relative person hour

rel. DD = relative decision density

rel. WT = relative weight priority

rel. CR = relative code reusability

rel. MC = relative module coupling

and w1 to w5 are weight factors which fall within 0 to 1 (excluding 0).

We have taken the relative value of all parameters i.e. individual value divided by the maximum parametric value. For example, if we obtained person -- hour values as 5, 7.4, 10 and 8 for different modules. We divide all the values by the maximum value i.e. 10 in this case. Thus our relative values will be 5 / 10 = 0.5, 7.4 / 10 = 0.74, 1.0 and 8 / 10 = 0.8 respectively. The advantage of using relative parameters is that it will fix the value of a particular parameter from 0 to 1. Thus no individual parameter will enjoy superiority over other.

We have added or subtracted these parameters and not multiplied or divided them because all the parameters are within the range 0 to 1. If we multiply them we will get

very small values. For example, a multiplication of 0.9 and 0.9 will lead to only 0.81 which is even lesser than both the values. We have to add 1 to the final value obtained so that MP value falls within the range of 0 to 5. If we don't add 1 to this value then MP value would vary from -1 to 4.

After calculating the MP values for all the modules, we partition them into 5 categories, namely Very High, High, Medium, Low and Very Low. This partition is in the descending order of the calculated MP values. To find out category ranges, we divide the difference between maximum and minimum MP values obtained, by the number of categories i.e. 5. Suppose that the maximum MP value is 4.50 and minimum MP is 2.00, the category range will be (4.50-2.00) / 5 = 0.5. Therefore category ranges are Highest 4.00 ≧ to ≦ 4.50, High 3.50 ≧ to < 4.00, Medium 3.00 ≧ to < 3.50, Low 2.50 ≧ to < 3.00 and Very low 2.00 ≧ to < 2.50.

In order to make tie breaks among the modules having same MP values, organization can decide the precedence list of the individual parameters. For example, it can fix decision density as the most important parameter and see if the modules with same MP values differ in this factor. If the values are same for this factor too then it can still go further into other factors till there is a tie break. If all the parametric values are found same then these modules should be kept in the same category.

If some modules cannot be tested without testing a particular module say M, then M should be given Very High priority irrespective of its MP value.

3.2 Proposed approach for testing

After prioritizing the modules in five categories, this paper attempts to find out maximum allowable cost and time for the testing. Since weight of cost and time can't be same [12]. We have used different stringency values for both of them. The deviation value varies for different organizations. A sample of stringency values is given in Table 1:

Table 1 Sample Stringency Values for Different Module Categories

Module Category	Percentage Stringency for Time	Percentage Stringency for Cost
Very High	25%	22%
High	20%	18%
Medium	15%	12%
Low	10%	7%
Very Low	4%	3%

Let T, C be the total time and cost available to release the software. Our aim is to the test all the modules within T and C. But if we are not able to do this then at least Very High, High and Medium ranged modules should be tested. We set the fault tolerance = 0 for the first time testing of all the modules of a particular category (e.g. Very High) and find out actual time and cost for testing.

Then we start testing and note down the actual time, cost and percentage of faults obtained for each category. Note that these values are obtained after complete testing of entire individual category. Afterward we find out the deviation from optimal testing time and optimal cost from (5) and (6).

Since, time and cost might have different weights depending upon organizational needs; we modify the formula for calculating δ. The new calculations for determining δ are given by (11):

$$\delta = m\alpha + n\beta \qquad (11)$$

Where m and n are constants which are determined by organization in such a manner that there sum turns out to be 1. These constants are useful in giving different weights to cost and time. Note that we are using (11) instead of (7) for calculating limiting factor.

Table 1 is used to compute the maximum value of δ for a given category. For example, with the sample stringency values given in the Table 1, δ_{max} for Very High category will be m (0.25) + n (0.22). If the values of m and n are 0.6 and 0.4 respectively then δ_{max} for Very High category is 0.6 (0.25) + 0.4 (0.22) = 0.24. Thus we can calculate δ_{max} for all the categories. The summary calculation is given in Table 2 (with m = 0.6 and n = 0.4).

Table 2 Sample δ_{max} for Different Module Categories

Module Category	δ_{max}
Very High	0.24
High	0.19
Medium	0.14
Low	0.09
Very Low	0.04

If the δ values obtained fall under the δ_{max} limits, we understand that this category had faults within the estimated fault limit so we move further to test lesser priority

modules. But, if the δ values exceed the estimated boundary, we calculate the faults obtained in the testing to compare it with fault tolerance. Now there can be 2 cases:

Case I. actual faults ≦ fault tolerance

In this situation, we suggest to proceed to the lesser priority components. If the faults are within tolerance range, then the remaining faults can be corrected at the maintenance time. Sometimes it may also happen that there are no faults in the tested category but still the observed testing time is higher. This is because of bad testing policy, inexperience of tester or some other environmental factor.

Case II. actual faults > fault tolerance

Here, we have to debug the modules based on the faults detected till either the observed faults are within tolerance or the resources are over. After testing each time, we increase the fault tolerance a little bit. It prevents us from getting stuck in an infinite loop of testing the same category again and again. We follow this approach because we want to test maximum number of modules within limited software release time. However our aim is to debug severe bugs residing in the software.

Initially we assume zero fault tolerance for first iteration. Then we calculate new fault tolerance for each iteration, as given in (12):

$$\text{new fault tolerance} = \text{Min}(\text{max\_tolerance}, \%\ \text{faults obtained in last iteration} - \text{min\_improvement}) \quad (12)$$

Where max_tolerance is maximum fault tolerance variable whose value increases by 2% and min_improvement is minimum fault improvement variable whose value increases linearly after each iteration. The maximum value of max_tolerance is 10%. After each iteration, we increase the value of max_tolerance because our resources are become crucial. On the other hand, we increase min_improvement because it ensures less number of faults than the previous iteration. The value of variables varies from organization to organization depending upon their needs.

For example, if an organization wants to permit less fault tolerance than the value of max_tolerance has to be less and min_improvement be higher.

For example, if there are 8% actual faults in a category, max_tolerance = 4% and min_improvement = 1% then new fault tolerance will be Min (4, 8 -1) = 4.

We repeatedly test a particular category and calculate new time and cost until the errors come in the fault tolerance limit, each time exceeding the fault tolerance. If the number of faults obtained in many consecutive iterations are same i.e. we do not get any further improvement even after much iteration, than management can decide to transfer this module to some other tester. At the end of T and C, we should be able to test Very

High, High and Moderate modules. If we are stuck in a particular category for a very large time and not able to finish these 3 categories then we need to report the managers that the software is error-prone and it is entirely their risk to launch the software in market.

It should be noted that even if we are not able to test Low and Very Low categories we prefer to launch the software.

The summary of the above suggested strategy is shown in the table 3.

Table 3 Summary of Suggested Actions Based on Fault Tolerance

$\delta <= \delta_{max}$	Actual fault <= Fault Tolerance	Suggested Action
Yes	No consideration	Move to lesser priority modules
No	Yes	Move to lesser priority modules
No	No	Increase fault tolerance, debug and test again to calculate new cost and time

Further work can be done in selecting the sample of test cases for low level categories when we don't have enough time and cost left to test these categories completely.

4. Case study

We applied the concept to a trie based dictionary software. A trie is an ordered tree data structure [16] used to store associative array. The position of the node in the tree showed only what key it was associated with [14]. In order to build a dictionary, each node contained a single character; words can be retrieved in pre-order traversal of the trie. It takes an input text file containing words and corresponding meanings and stores them in a trie. After this it performed various dictionary operations. The major functionalities of this software product could be viewed as:

- Main menu driven option module for user input
- Reading a file to store data
- Finding a word
- Inserting a new word to dictionary
- Adding meaning to the word
- Editing the word meaning

- Storing back the dictionary to a new file

We identified seven major modules namely main module, read file, find word, insert word, add meaning, edit meaning and print. The weight factor for these modules could be calculated as given in Table 4, where relative weight is total/max value in total column:

Table 4 Calaulating Weight Factor for Dictionary Modules

Module Name	Requirement Factor (0.6) (0.6 is weighted factor)	Risk Factor (0.4) (0.4 is weighted factor)	Total	Relative Weight
Main	7	7	7	0.78
Read File	7	8	7.4	0.82
Find Word	9	7	8.2	0.91
Insert Word	9	9	9	1
Add Meaning	7	8	7.4	0.82
Edit Meaning	6	7	6.4	0.71
Print	9	6	7.8	0.87

The, main module had requirements rank as 7 and risk rank as 7, so its total rank was 0.6(7) + 0.4(7) = 7. This ranking i.e. 7 for requirement and 7 for risk factor was given by developers, managers and customers based on their experience [10]. To calculate relative weight, we divide total by max (total). In this case max (total) was 9 that cames from Insert Word module. So, the relative weight for Main module was 7 / 9 = 0.78. Similarly, other modules' relative weight could be calculated.

Then we found out coupling among these modules, calculations for coupling are shown in the Table 5. The procedure to find out relative coupling in the Table 5 is coupling/max (coupling).

Table 5 Calaulating Coupling for Dictionary Modules

Module	Module Coupling = 1 /(input, output, global, fan out, fan in)	Coupling =1- 1/M	Relative Coupling
Main	1 / 0+1+4+6+0	0.91	1
Read File	1 / 2+0+1+1+1	0.80	0.88
Find Word	1 / 2+1+0+0+1	0.75	0.83

Table 5 Calaulating Coupling for Dictionary Modules (Continued)

Module	Module Coupling = 1 /(input, output, global, fan out, fan in)	Coupling =1- 1/M	Relative Coupling
Insert Word	1 / 4+2+0+2+2	0.90	0.99
Add Meaning	1 / 3+1+1+0+2	0.86	0.94
Edit Meaning	1 / 3+1+1+0+1	0.83	0.91
Print	1 / 3+0+1+1+1	0.83	0.91

Similarly we computed rest of the parameters. Final values for the modules are given in the Table 6.

Table 6 Final Modules Priority Values

Module	A	B	C	D	E	F	G	MP	Category
Main	0.67	36	183	0.74	0.78	0	1	4.19	High
Read File	0.17	4	31	0.48	0.82	0	0.88	3.35	Low
Find Word	0.5	5	23	0.81	0.91	0	0.83	4.05	High
Insert Word	1	15	92	0.59	1	0	0.99	4.58	Highest
Add Meaning	0.17	4	26	0.56	0.82	0	0.94	3.49	Medium
Edit Meaning	0.17	4	37	0.41	0.71	0	0.91	3.20	Very Low
Print	0.83	7	26	1	0.87	0	0.91	4.61	Highest

Where a = relative person hour

b = total cyclomatic complexity

c = logical lines of code

d = relative decision density

e = relative weight factor

f = relative code reusability

g = relative coupling

MP = module priority which is calculated by (10)

Here, we saw that the maximum MP value was 4.61 and minimum MP was 3.20. So, the category range was (4.61-3.20) / 5 = 0.28. Hence category range Highest was in between 4.33 ≧ to ≦ 4.61, High is in between 4.05 ≧ to < 4.33 and so on. In our case study, though last 3 modules fell in the Medium category yet different categories were assigned to them i.e. 3.49 in Medium, 3.35 in Low and 3.20 in Very Low, just for demonstration purpose, as our project size was not large enough. But in large projects, each category will have some modules in it.

After this we calculated δ_{max} values for these modules from (11) and using stringency values from Table 1. We also found out optimal time and cost from (2) and (3) respectively. Optimal values are given in Table 7. Please note that for different values of c1, c2, c3, t and T these values are different.

Table 7 Optimal Cost and Time Values

Category	Optimal time	Optimal cost	Max time	Max cost
Highest	5.00	10.50	6.25	12.81
High	4.50	8.80	5.40	10.38
Medium	3.00	6.90	3.45	7.73
Low	3.00	5.80	3.30	6.21
Very Low	2.19	5.20	2.28	5.36

Then, assuming that we had total 18 time units and 36 cost units to test the software (you can calculate from cocomo model [11] also). We will calculate the highest priority category modules first. After testing this category we obtained the actual time, cost and percentage fault values as 5.5, 11 and 2% respectively. Since here $\delta < \delta max$ so we moved to the next priority modules i.e. high priority modules in spite of the faults exceeding the tolerance limit.

Now we received time, cost and percentage fault values as 6, 10 and 3% respectively. Here $\delta = (6 – 4.5) / 4.5 * 0.6 + (10 – 8.8) / 8.8 * 0.4 = 0.25$ from (5), (6) and (11). Since $\delta > \delta_{max}$ as the δ_{max} value for high priority modules was 0.19 from Table 2, we considered percentage of faults obtained. Since this percentage fault was outside the fault tolerance (for first iteration it is 0), we had to test this category again with an increased fault tolerance of 2% (if max_tolerance = 2% and min_improvement = 1% then from (12) new tolerance = min (2, 3-1)). Next time, we got time, cost and percentage faults as 4.7, 6.7 and 1% respectively. Again here $\delta < \delta max$ so we moved to lesser priority i.e. medium priority modules.

This time we got time, cost and percentage faults as 3, 8 and 0% respectively. Since $\delta > \delta max$ in this case, we had to look into percentage faults. Since these faults were within tolerance, we did not look into this priority again.

By that time, our resources (time and cost) were over therefore we checked which priority modules were tested enough. Since we could test highest, high and medium priorities, the software was ready to be launched.

5. Conclusion

This paper has illustrated how we can prioritize the software modules in order to test the important modules primarily. Non homogenous poisson processed model helps us to calculate the optimal testing time and cost. After allowing a little deviation from these values and accepting low risk faults in the system, we can test the software effectively even if we have limited resources available with us. Fault tolerance concept assists us to test the software in a given time and cost. At the end of the resources, we can also find out if we have tested enough or there is a further need of testing the important modules. This facilitates us to make a decision whether the software product is ready to be released in the market or not. The approach described in this paper is more suitable for the situations where there are fixed values for testing time and cost. Further research can be carried out to find out more accurate ways of assigning categories to the modules based on clustering or graph theory. Yet another open issue is to determine the appropriate values for min_improvement and max_tolerance variables.

Acknowledgement

We thank to Dr. Sanjiv K. Choudhary, BITS Pilani for many insightful discussions for proof reading of this article.

References

1. Arika, K.O, Tsai, W.T., Poonawala, M. and Suganuma H. (1988) 'Regression testing in an industrial environment', *Communications of ACM*, Vol. 41, No. 5, pp. 81-86.

2. McDaid, K. and Wilson, S.P. (2001) 'Deciding how long to test software', *Journal of the royal Statistical Society: Series D (The Statistician)*, Vol. 50, No. 2, pp. 117-134.

3. Musa, J.D. and Ackerman, A.F. (1989) 'Quantifying software validation: When to stop testing', *IEEE Software*, Vol. 6, No. 3, pp. 19-27.

4. Ehrlich, W., Prasanna, B., Stampfel, J. and Wu, J. (1993) 'Determining the cost of a stop-test decision', *IEEE Software*, Vol. 10, No. 2, pp. 33-42.

5. Goel, A.L. and Okumoto, K (1979) 'A time dependent error detection rate model for software reliability and other performance measures', *IEEE Transactions on Reliability*, Vol. R-28, No. 3, pp. 206-211.

6. Goel, A.L. and Okumoto, K. (1980) 'A time dependent error detection rate model for software performance assessment with applications', *Proceedings of National Computer Conference*, RADC-TR-80-179.

7. Goel, A.L. and Okumoto, K. (1981) 'When to stop testing and start using software?', *Proceedings of ACM*, Vol. 10, No. 1, pp. 131-138.

8. Srivastava, P.R., Pareek, D., Sati, K., Pujari, D.C. and Raghurama, G. (2008) 'Non homogenous poisson process based cumulative priority model for determining optimal software testing period', *ACM SIGSOFT Software Engineering Notes*, Vol. 33, No. 2.

9. Jones C. *(1991) Applied Software Measurement*, McGraw-Hill, New York.

10. Srivastava, P.R., Kumar, K., and Raghurama, G. (2008) 'Test case prioritization based on requirements and risk factors', *ACM SIGSOFT Software Engineering Notes*, Vol. 33, No. 4.

11. Pressman, R.S. (2005) *Software Engineering: A Practitioner's Approach*, McGraw-Hill.

12. Srivastava, P.R. (2008) 'Model for optimizing software testing period using non homogenous poisson process based on cumulative test case prioritization', *Proceedings of the IEEE TENCON*, Hyderabad, India, pp. 18-21.

13. Srivastava, P.R. and Pareek D., 'Component Prioritization Schema for Achieving Maximum Time and Cost Benefits from Software Testing', *Information System, Technology and Management, Communication in Computer and Information Science Series*, Springer Berlin Heidelberg, Vol. 31, pp. 347-349.

14. Trie. http://en.wikipedia.org/wiki/Trie

15. Mathur, A.P. (2008) *Foundations of software testing*, Pearson Education, India.

16. Goodrich, M.T., Tamassia, R. and Mount, *D.M. (2004) Data Structures and Algorithms in C++*, John Wiley & Sons Ltd., New York.

About the authors

Praveen Ranjan Srivastava is working in computer science and information systems group at Birla Institute of Technology and Science (BITS), Pilani India. He is currently doing research in the area of Software Testing. His research areas are software testing, quality assurance, testing effort, software release, test data generation, agent oriented software testing, stopping testing, soft computing techniques. He has a number of publications in the area of software testing. Contact him at praveenrsrivastava@gmail.com

Chetan Mittal, Ajay Rungta, Vaibhav Malhotra are all presently doing M.E (Software Systems) in Computer Science and Information Systems Group, at Birla Institute of Technology and Science, Pilani, India.