Measuring and Assessing Staffing Level, Cost Analysis for Debugging Activities Using Queuing Theory

Sangeetha. M[1], Arumugam. C[2], Senthil Kumar K. M[3]

[1] Coimbatore Institute Of Technology, Coimbatore, India
[2]sri Ranganather Instutute Of Technology, Coimbatore, India
Email: kmsenthildrkumar@gmail.com

Abstract: Reliability is the probability of a software working correctly over a specific period of time. Reliability predictions and assessments are important in ensuring the quality. Many approaches has been given like rate based approaches for the reliability of the software and to analyze the reasons for the failure of the software. Criteria for the reliability of the software, number of debuggers or developers available are not taken into account. Newly detected faults have to wait for some time since all the debuggers will be busy in detecting the faults which they found previously. Time taken to remove the fault is taken into consideration and the main fact relies in it is that less number of faults been removed when compared to the number of faults detected. This is mainly because fault detection is continued as faults are also removed side by side. Taking the previous out comings into consideration, our project proposes a rate-based simulation method by applying the queuing theory for the debugging behavior during the development of the software. G/G/∞ and G/G/m models have been used in our proposed method. This method is used for the real software failure. This approach helps to predict the debuggers' performance and the cost effectiveness.

[Sangeetha. M, Arumugam. C, Senthil Kumar K.M. **Measuring and Assessing Staffing Level, Cost Analysis for Debugging Activities Using Queuing Theory.** *Life Sci J* 2013:10(1):3252-3258]. (ISSN: 1097-8135). http://www.lifesciencesite.com. 410

Keywords: Fault correction, non-homogeneous Poisson process (NHPP), software reliability growth model (SRGM), software testing

INTRODUCTION

Software should meet requirements specifications. Best quality software satisfies the need of the customer. Historically, the word "quality" has been adapted and has evolved together with the different technologies to which it has been applied. Each software should not contain compliance. It implies loss of quality or less trust on product. Inspection process goal was to avoid corrections through the identification of product deviations from requirement specification [2].

Software metrics can be classified into three categories: product metrics, process metrics, and project metrics. Product metrics describe the characteristics of the product such as size, complexity, design features, performance, and quality level Software industry focuses on the following principles: 1. Software requirements are the quality metric fundamental. Lack of compliance with requirements is a quality failure. 2. Standards establish development criteria. Absence of standards means, in many cases, low quality [5]. 3. Indirect measures (e.g. usability, maintainability, etc.) and direct measures (e.g. lines of code).

Software can also have small unnoticeable errors or drifts that can culminate into a disaster. On February 25, 1991, during the Golf War, the chopping error that missed 0.000000095 second in precision in every 10th of a second, accumulating for 100 hours, made the Patriot missile fail to intercept a scud missile. 28 lives were lost.

Fixing problems may not necessarily make the software more reliable. On the contrary, new serious problems may arise. In 1991, after changing three lines of code in a signaling program which contains millions lines of code, the local telephone systems in California and along the Eastern seaboard came to a stop.

In recent decades, rate-based simulation approaches have been proposed to analyze stochastic failure processes [2], [7], [11]. Simulation approaches relax certain unreasonable assumptions which are common in model-based approaches. This type of approach can also extend the reliability process to encompass the entire software life-cycle [11]. However, we found that most of the research on simulation approaches has not considered the limitations of debugging resources during the fault correction process (FCP). In fact, for project managers, such kinds of information will be valuable, and helpful. Musa [3] reported that the number of debugging personnel is one of the major constraints on the rate of testing. Therefore, in this paper, we will incorporate the queuing model into the rate-based simulation framework to approximate reality more closely.

Through the proposed simulation framework, possible debugging behavior will be analyzed and discussed under consideration of the debugging team size.

The remainder of this paper is organized as follows. Section II reviews existing methods for software reliability prediction. Then we will propose two rate-based simulation procedures to analyze both the FDP, and FCP in Section III. In the proposed framework, debugging behavior is analyzed based on the concept of queuing theory. In Section IV, experiments based on two real data sets are discussed in terms of performance, and cost-effectiveness. Finally, Section V concludes the paper.

SOFTWARE RELIABILITY PREDICTION APPROACHES

Farr & Lyu [2] also pointed out that the NHPP model has formed the basis for the models using the observed number of faults per unit time group. However, we observed that most of these models deal solely with FDP [17]. In reality, the terms testing and debugging are related but distinct. Testing is the process of exercising a program with the intention of revealing inherent faults, while debugging activity localizes the root cause of the detected fault, and then corrects the fault [18]. Because fault removal may require time and effort, the number of removed faults will lag behind the total number of detected faults. Musa [3] argued that the fault removal process is characterized on an average basis by assuming that the fault correction rate is proportional to the hazard rate. He called the proportionality constant a fault reduction factor. In addition, Wood [19] reported that instantaneous repair is not realistic in practice. Therefore, model-based methods should be modified to take into account the FCP [17].

Apart from the NHPP models, many researchers have applied neural networks to predict FCP, and to estimate software reliability [6], [12], [13]. Karunanithi & Malaiya [12] proposed neural network architecture which first accepts the execution time as the input, and then shows the number of detected faults. This type of framework models the software's reliability using different neural networks, such as a recurrent neural network. Another kind of neural network framework models software reliability based on a multiple-delayed input/single-output neural-network architecture. For example, Cai et al. [12] designated the most recent 50 inter-failure times as the multiple-delayed inputs to forecast the occurrence of the next failure. Similarly, Tian & Noore [3] proposed an evolutionary neural-network modeling approach for the prediction of cumulative failure time based on this architecture. Su & Huang [6] also proposed an ANN-based dynamic weighted combinational approach to predict software reliability [11] reported that instantaneous repair is not realistic in practice. Therefore, model-based methods should be modified to take into account the FCP [7].

Apart from the NHPP models, many researchers have applied neural networks to predict FCP, and to estimate software reliability [6], [11]. Karunanithi & Malaiya [2] proposed a neural network architecture which first accepts the execution time as the input, and then shows the number of detected faults. This type of framework models the software's reliability using different neural networks, such as a recurrent neural network. Another kind of neural network framework models software reliability based on a multiple-delayed input/single-output neural-network architecture. For example, Cai et al. [2] designated the most recent 50 inter-failure times as the multiple-delayed inputs to forecast the occurrence of the next failure. Similarly, Tian & Noore [3] proposed an evolutionary neural-network modeling approach for the prediction of cumulative failure time based on this architecture. Su & Huang [6] also proposed an ANN-based dynamic weighted combinational approach to predict software reliability. Besides, Hu et al. [3] further studied a major ANN architecture, the Elman recurrent networks, to model both the FDP, and FCP for software reliability analysis.

PROPOSED APPROACH

The simulation algorithm can be applied to each individual activity during SDLC. Later, Gokhale & Lyu [7] proposed a simulation technique to analyze structure-based software reliability. They believed that the time required by fault repair should be considered explicitly. They also extended the simulation to the reliability assessment on the application level. Recently, Gokhale et further considered the possibility of imperfect debugging in the simulation approach. However, we found that existing published simulation techniques seldom consider the limitations of debugging resources, and this oversight may not be reasonable. In practice, the number of qualified debuggers will be controlled during SDLC. In the next section, we will apply queuing theory to mo del software fault correction activities through simulation procedures.

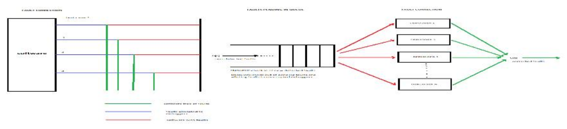

Figure 1. Software testing and debugging activities The software system is subject to failures at random times caused by the manifestation of the remaining faults in the system.

2) All faults are independent, and equally detectable. The probability that a failure will be experienced during (t, t+Δt) is Λ(t) Δt approximately, and the probability that two or more failures will occur during (t, t+Δt) is negligible.
3) The correction of faults takes non-negligible time, i.e., explicit repair. The probability that a fault is corrected in time interval (ts, ts+Δt) is μ * Δt. Further, fault removals do not affect the ongoing activities of fault detection.
4) No new faults are introduced during the correction process.
5) Available, and qualified debuggers are always sufficient.

The debugging system is modeled by a queue system (G/G/∞). Each time a failure occurs, there is no lag to allot a debugger to the detected fault.

Based on these assumptions, Procedure #1 was developed, and is depicted in Fig. 2. Procedure #1 accepts two parameters as inputs: the total number of execution time units, defined as stop_time; and the consumed time of each run, denoted by dt. The length of each time unit should be consistent with the failure data collection form. Further, each time unit is divided into a large number of runs, and the length of each run should be short enough that multiple events in a run are rare [2], [7]. That is, the variations of failure rate in (t, t+dt) should be insignificant. In addition to the two inputs, certain variables are also used in the simulation to represent the major components of the debugging system, and to gather useful statistics. The variable current_time represents a clock, which also indicates the cumulative execution time to the present. The array correction, each element of which contains a fault_info, is used to keep track of the status of each fault. Further, working_server denotes the current number of busy debuggers, while max_server logs the number of utilized debuggers at peak time. Finally, cumulative_arrival and cumulative_departure are integers used to count the numbers of cumulative detected faults and cumulative removed faults, respectively.

There are several ways to derive the simulation. Our framework adopts a random-number generator, which is common. Following mathematical probability distributions, the generator is programmed to generate arrivals, departures,
and so on. During simulation, actions taken in each run consist of two steps: detecting, and correcting.Detecting: Following similar work in [4], and [7], we can simulate the FDP. At the outset of each run, the function occur() will be invoked to determine whether the testers detect a fault in this run.

This means that the testers may detect a fault if is greater than Λ(t). Once the occur function returns 1, is increased, the value of is updated, and the state of the detected fault will be recorded. Lines 10–14 in Procedure #1 show the activities taking place as a result of each failure occurrence.

Correcting: Departing from the detection step, we commence diagnosing the status of each detected fault by checking all elements of the array correction. If an open-remaining fault (a detected but uncorrected fault) is found, the function leave(ts) determines whether this fault will be corrected in this run. Similar to the occur function, the success of fault removal relies upon the comparison between dt, and the random number. If dt, then the dedicated debugger successfully corrects this fault in this run. The necessary actions taken due to this successful repair are given in Lines 19–22. Otherwise, this fault cannot be corrected at this time, and will be reexamined in the next run. From Assumption 3), we know that the necessary correction time is non-negligible. Hence, the return of leave(0) is given as 0, ensuring that the fault detected in the current run will not be removed immediately.

B. Procedure 2

In Procedure #2, each run consists of three steps: debugger allocation, fault detection, and fault correction.

Allocation: This step allocates the qualified debuggers to the faults pending in the queue. First, we check the existence of unoccupied debuggers by comparing with. If all debuggers are not busy, a pending fault will be deleted from the waiting queue, and will occupy one debugger. The activities for debugger assignment are shown in Lines 11–15, which will be repeated until available debuggers are exhausted, or the waiting queue becomes empty.

Detecting: Once a fault is detected, we first increase, and then determine whether available debuggers exist. If (meaning that some debuggers are still available), the actions are the same as those taken during the detection step of Procedure #1. Otherwise, the detected fault will be inserted into the queue to await service. Related activities are given in Lines 19-29 of Fig. 3.

Correcting: Entering this step, the status of each fault will first be checked. If there are faults being repaired, the function leave will be executed to determine whether the repairs will be successful. Following each successful repair, the actions described in Lines 35–38 will be taken, i.e., an occupied debugger will be released. Conversely, faults which are not corrected in the current run will be reexamined in the next run.

The above three steps will be reiterated prior to the. In addition to the information obtained through Procedure #1, we further have the average time spent in the waiting queue, the average length of the waiting queue, and other statistics. Although productive research has used neural network approaches to predict the increment of the failure process, most training algorithms for neural network approaches suffer from the overfitting problem. That is, the fitting bias of the training set is very slight regarding known data, but the bias is unpredictably large when new data are presented to the network [3]. Determination of the proper number of neurons is another common problem in the field of neural network research . Moreover, Tausworthe & Lyu [2],argued that most SRGM only focus on the failure observation during the test phase, or the operational phase. They reported that the assumptions of most SRGM lead to the over-simplification of the failure process. Thus, general simulation techniques have been developed to relax certain unreasonable assumptions [2].

For ease of discussion, we let ≥ 0} be the stochastic failure process that represents the number of failures observed in an execution interval (0,). If the failure behavior is modeled by a failure rate, N(t) can be modeled by a class of NHCTMC [7]. That is, the behavior of the stochastic process N(t) purely depends on the rate function for each state of the software system. If the state is represented by the number of occurrences of the event, it is known as a pure birth NHCTMC.

```
        void    Simulation_Procedure    (double
stop_time, double dt)
  {
    double current_time = 0;
    int working_server = 0, max_server = 0;
    struct fault_info correction[Max_Size];
    int       cumulative_arrival       =       0,
 cumulative_departure = 0;
    while ( current_time < stop_time) {
        DETECTING:
        if( occur() ){
         working_server++;
           if (working_server > max_server)
             max_server = working_server;

 correction[cumulative_arrival].arrival_time       =
 current_time;
           correction[cumulative_arrival++].state =
 CORRECTING;
           }
        CORRECTING:
        for(int i = 0; i < cumulative_arrival; i++) {
             if        (correction[i].state         ==
 CORRECTING      &&      leave(current_time     -
 correction[i].arrival_time)) {
                        working_server--;
 correction[i].departure_time = current_time'
                          correction[i].state        =
 CORRECTED;

 cumulative_departure++;
                  }
           }
         current_time += dt;
 }
```

Figure. 2. Procedure #1

SOFTWARE RELEASE STRATEGIES AND COST ESTIMATIONS

In the high-technology market, the life cycle of software products may be so short that the manager is actually willing to deliver the software product with uncorrected faults by the scheduled deadline. Nevertheless, delivering a bad product may lead to customer dissatisfaction, and then cause damage to a software company's reputation [3]. Therefore, if some open-remaining faults still exist with the scheduled dead- line approaching, we assume that there are two debugging strategies to manage the project: Release Strategy A, which strictly enforces the hard-deadline, and delivers the software product with open-remaining faults; and Release Strategy B, which extends the deadline, and continues the fault correction until fixing all open-remaining faults. Based on these two strategies, we will study the expected cost and penalty of open-remaining faults as the project manager staffs the debugging team with different amounts of personnel. To simplify our analysis, we will only focus on those factors related to the staffing level of the debugging team. The factors common in both strategies will be ignored, such as the cost of discovering faults, and the penalties of faults which are not discovered before release.

Correcting: Entering this step, the status of each fault will first be checked. If there are faults being repaired, the function leave will be executed to determine whether the repairs will be successful. Following each successful repair, the actions described in Lines 35–38 will be taken, i.e., an occupied debugger will be released. Conversely, faults which are not corrected in the current run will be reexamined in the next run.

Release Strategy A: In addition to customer dissatisfaction, the penalties of remaining faults should also include the cost of fixing faults after release. The cost of fixing a fault after release is usually an order of magnitude greater than fixing the fault prior to release [4]. The cost function can be given as: Intuitively, there may be a negative relationship between the penalties of remaining faults, and the number of debuggers. But the amount of debuggers' salaries is proportional to the number of debuggers. To determine a suitable staffing level, and to re- duce the total cost, software project managers should strike a balance regarding the total debugger's salaries before release, and the penalties of open-remaining bugs after release.

```
void Simulation_Procedure (double stop_time, double dt, int staffing_level)
{
    double current_time = 0;
    int working_server = 0;
    struct fault_info correction[Max_Size], waiting_queue{max_Size];
    int num_correction = 0,cumulative_arrival = 0, cumulative_departure =0;
    int queue_head = 0, queue_tail = 0;
    while (current_time < stop_time) {
        ALLOCATION:
        while (working_server < staffing_level && queue_head!=queue_tail) {
            waiting_queue[queue_head].state = OUT_OF_QUEUE;
            waiting_queue[queue_head++].departure_time = current_time;
            correction[num_correction].state = CORRECTING;
        }
        DETECTING:
        if(occur()){
            cumulative_arrival++;
            if (working_server >= staffing_level){
                waiting_queue[queue_tail].state = ENQUEUE;
                waiting_queue[queue_tail++].arrival_time = current_time;
            }
            else {
                working_server++;
                if (working_server > max_server)
                    max_server = working_server;
                correction[num_correction].arrival_time = current_time;
                correction[num_correction++].state = CORRECTING;
            }
        }
        CORRECTING:
        for (int i=0; i<num_correction; i++) {
            if (correction[i].state == CORRECTING && leave(current_time - correction[i].arrival_time)) {
                working_server--;
                correction[i].departure_time = current_time;
                correction[i].state = CORRECTED;
                cumulative_departure++;
            }
        }
        current_time += dt;
    }
```

Figure. 3. Procedure #2

Cost1 = debuggers salaries of + cost of fixing faults after release + penalty customer (6)

Release Strategy B:

If the debugging activities are continued until all detected faults are removed, the cost will exclude the penalties caused by the open-remaining faults after release, but the scheduled project deadline, and software release could be extended accordingly. Thus, in addition to extra debuggers' salaries during the extended period, the penalty of the declining market position is inevitable. The expected cost for this strategy can be calculated by

Cost2 =original debuggers salaries +penalty of lost market position+Extra debugger's salaries during extended period. (7)

Similarly, to minimize the expected cost, it is necessary to analyze the trade-off between two costs: the debugger's salaries, and the penalties due to late release.

RESULTS

The data set was from system T1 of the Remote Air Development Center project.The failure data were carefully collected under strict supervision. System T1 was applied to real time command and control, including 21,700 delivered object instructions. Over the course of 21 weeks, 9 programmers detected and removed 136 faults.

1. The average of fault removals per week.
2. The faults detected during the period of 21 weeks.
3. The average measure of all faults. Note that the waiting time means the time of the fault pending in waiting queue, and the response time indicates the total time spent in the queuing system. Besides, because some detected faults may not be removed yet at the end of 21 weeks, the simulation of correction processes, and the statistics are continued until all pending faults are addressed.

Figure. 4. Datas of Remote Air Development Center

Limitation of Available Debuggers	Throughput (by the end of 21 weeks)		Time to remove faults (weeks)2,3	Avg. waiting time (weeks)3	Avg. response time (weeks)3	Avg. queue lgth3	Debugger utilization (%)
	Open-remaining faults	Avg. removals					
Unlimited	2	6.52	22	0.00	0.66	0.00	-
12	2	6.52	22	2.45 * 10-3	0.67	1.55*10-2	36.79
11	2	6.52	22	6.69 * 10-3	0.67	4.22*10-2	40.13
10	2	6.52	22	1.37 * 10-2	0.68	8.64*10-2	44.14
9	2	6.52	22	3.21 * 10-2	0.70	0.20	49.05
8	2	6.52	22	0.10	0.77	0.65	55.18
7	5	6.38	22	0.37	1.03	2.32	62.75
6	19	5.71	24	1.03	1.69	5.97	66.15
5	36	4.90	26	2.15	2.81	11.47	68.61
4	53	4.10	30	4.02	4.69	18.64	71.15
3	71	3.24	38	7.44	8.11	27.23	73.75
2	95	2.10	52	14.23	15.28	39.08	77.26
1	113	1.24	97	36.79	37.45	52.72	84.67

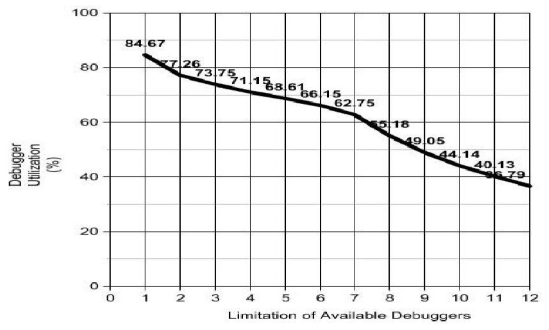

Figure 5. Performance comparisons between different staffing levels as the value of varies from 1 to 8

According to the above analyses, the limitation on the debugging team size is not the bottleneck to enhance performance when the number of personnel is more than 7. Increasing the number of debuggers can only increase the number of simultaneous working debuggers, but it does not reduce the consumed time taken for a debugger to fix a fault. Therefore, when the staffing level can bear the load, more debuggers cannot improve the throughput. If the manager wants to ameliorate the performance, it is necessary to improve the debuggers' skills, i.e. to increase the value. Fig.4 shows some performance comparisons between different staffing levels as the value of varies from 1 to 8. Due to space limitations, we only demonstrate the number of open-remaining faults, and the time to remove all faults in each condition. As is clear from Fig. 7, both statistics decrease with the growth of , and the staffing. The utilization of 6 debuggers is very low at the beginning. However, because the number of detected faults grows rapidly from the 8th week to the 18th week, 6 personnel seem unable to bear the load. Re-staffing may be reasonable in both conditions. Using the pro- posed framework, project managers can easily estimate the influence caused by re-staffing the debugging system, and further decide whether to make an adjustment.

CONCLUSION

In this paper, we modeled debugging behavior using queuing models. Two simulation procedures were developed to simulate the stochastic FDP, and FCP under different conditions. The proposed framework can help to understand/infer the current/future situations of the on-going project, or to reconstruct the possible behavior of the completed project. The applications of the pro- posed procedures are illustrated through two real data sets. The case studies show that the proposed simulation procedures can analyze the influence on the performance, and the cost related to software debugging when the number of allocated debuggers changes. This useful, important information can guide project managers in the estimation and adjustment of the staffing needs for debugging systems. Further, the proposed procedures are also useful when the project is planned using the techniques of expert judgment, or estimation by analogy.

ACKNOWLEDGEMENTS

I would like to thank Prof. K.M. SenthilKumar for providing support and material related to educational research. Also, I am grateful to my Principal and colleagues for their support in teaching practical sessions of the software engineering module in addition to their valuable feedback as tutors. Last, but not least, I am grateful to Coimbatore Institute of technology for the early experience in teaching this subject, and especially to Dr. C.Arumugam for the fruitful discussions about this type of research.

REFERENCES

[1] Dwyer, D. & D'Onofrio, P., (2011). Improvements in estimating software reliability from growth test data. Reliability and Maintainability Symposium (RAMS), 2011 Proceedings - Annual, 1 – 5.

[2] Goel, A. L. & Yang, K. J. (1997). Software reliability And readiness assessment based on the non-homogenous Poisson process, Advances in Computers, 45, 197-267.

[3] Gokhale, S.S., Lyu, M.R. & Trivedi, K.S. (2006). Incorporating fault debugging activities into software reliability models: a simulation approach. Reliability, IEEE Transactions on, 55(2), 281 – 292.

[4] Hung, C. Y., Lyu, M. R. & Kuo, S. Y. (2003). A unified scheme of some non-homogeneous poisson process models for software reliability estimation, IEEE Trans. On Software Engineering 29(3), 261-269.

[5] Hung, C. Y. & Lin, C. T.(2006).Software reliability analysis by considering fault dependency and debugging time lag, IEEE Trans. on Reliability, 55(3), 436-450.

[6] Hung, C. Y., Lin, C. T., Kuo, S. Y., Lyu, M. R. & Sue, C. C. (2004). Software reliability growth models incorporating fault dependency with various debugging time lags,Proceedings of the 28th Annual International Computer Software and Application Conference, Hong Kong, China, 186-191.

[7] Hung, C. Y., Lin, C. T., Lo, J. H. & Sue, C. C. (2004). Effect of fault dependency and debugging time lag on software error models, Proceedings of the 2004 IEEE Region 10 Conference, Thailand, 243-246.

[8] Kapur, P.K., Pham, H., Anand, S. & Yadav, K. (2011). A Unified Approach for Developing Software Reliability Growth Models in the Presence of Imperfect Debugging and Error Generation Reliability, IEEE Transactions on , 60 (1), 331 – 340.

[9] Lyu, M. R. (1993) .Handbook of Software Reliability Engineering: McGraw-Hill. 428-443. Ohba, M. (1984).

[10] H. Okamura, H. Furumura, and T. Dohi, “On the effect of fault re- moval in software testing-Bayesian reliability estimation approach,” in Proceedings of the 17th International Symposium on Software Reliability Engineering, Raleigh, North Carolina, USA, November 2006, pp.247–255.

[11] N. Karunanithi and Y. K. Malaiya, “Prediction of software reliability using connectionist models,” IEEE Trans. Software Engineering, vol. 18, no. 7, pp. 563–574, July 1992.

3/3/2013