

RESEARCH ARTICLE

Queueing-Based Simulation for Software Reliability Analysis

JHIH-SIN LIN¹ AND CHIN-YU HUANG², (Member, IEEE)¹Silicon Motion, Inc., Zhubei, Hsinchu 302082, Taiwan²Department of Computer Science, National Tsing Hua University, Hsinchu 300044, Taiwan

Corresponding author: Chin-Yu Huang (cyhuang@cs.nthu.edu.tw)

This work was supported by the Ministry of Science and Technology, Taiwan, under Grant MOST 110-2221-E-007-035-MY3 and Grant MOST 111-2221-E-007-079-MY3.

ABSTRACT As modern software system is growing in size and complexity, the customer expectations for software quality have become higher. In the past, many software reliability growth models (SRGMs) were proposed and they are helped to evaluate the quality of developed software. It is worth noting that some of SRGMs can be used to model the fault detection process (FDP) and the fault correction process (FCP) through an infinite server queueing (ISQ) system or a finite server queueing (FSQ) system. However, it can also be found that most ISQ and FSQ models were developed on a first come first served basis. In this paper, we propose to use the queueing-based simulations to describe the behavior of FCP and assess the software reliability instead of using model-based approaches. Our proposed queueing-based simulation techniques and simulation procedures will be able to thoroughly investigate the FCP and easily provide system performance information estimated based on the staffing level, the average response time, and the average waiting time. Numerical examples based on three real failure data are given and discussed. Our experiments show that the proposed simulation procedures obtain a good prediction capability for software reliability. We expect that the proposed methods can provide effective information for software developing management and help decision makers in resource allocation and cost control.


INDEX TERMS Fault detection, fault correction, software reliability growth models (SRGMs), preemptive priority queue, rate-based simulation.

I. INTRODUCTION

In modern society, software is already an indispensable part of our lives. To assess software quality, ISO/IEC 9126 is a useful reference [1]. The quality model specifies six characteristics including *Functionality*, *Reliability*, *Usability*, *Efficiency*, *Maintainability*, and *Portability*. Among these characteristics, *reliability* is generally regarded as a key factor in software quality evaluation. Presently, many software reliability growth models (SRGMs) have been published and are used to evaluate the quality of developed software. Reference [2], [3], [4], [5], [6], [7], [8], [9], [10]. In general, SRGMs are formulated in terms of random processes [3]. SRGMs can typically be used to help managers truly understand the current project status. But Musa [5] once argued

that many research studies of the software reliability modeling that follow are either theoretical or based on small-scale projects or limited failure data, so they must be generally viewed as heuristic and not yet ready for real-world use.

In the past, some of SRGMs used the infinite server queueing (ISQ) or finite server queueing (FSQ) systems to model the fault detection process (FDP) and the fault correction process (FCP) [11], [12], [13], [14], [15]. It can also be found that most ISQ and FSQ models were developed on a First-Come-First-Served (FCFS) basis. But practically, high priority faults should be fixed quickly to minimize their impact on software development and testing. But it also has to be noted that the fault correction time should not be ignored. Mockus et al. [16] reported that the high-priority faults should be corrected faster than the low-priority faults. Moreover, Zhang et al. [17] analyzed close source software and collected

The associate editor coordinating the review of this manuscript and approving it for publication was Claudia Raibulet .

its failure data. Their statistics demonstrated that bugs with the higher priority are fixed faster, which means that the bug-fixing time is shorter.

Priority scheduling [18], [19] is an operating system process scheduling algorithm and can be generally classified as preemptive or non-preemptive. Based on this concept, Lin and Huang once [9] proposed a preemptive priority queueing (PPQ) model that considers both a finite number of debuggers and different priority levels. In actuality, in addition to above model-based approaches, the discrete-event simulation offers an alternative to analytical models as it can represent the impact of different strategies that may be used during testing [4]. For example, rate-based simulation approach was proposed to relax certain unreasonable assumptions that are common in model-based approaches [4].

Based on our past studies [9], in this paper we further propose queueing-based simulation techniques and simulation procedures to analyze the behavior and reliability of the PPQ model. The proposed simulator can describe the process of fault correction in more detail and optimize various system parameters such as staffing level, average response time, and average waiting time. Some experiments based on real open source software (OSS) and closed source software (CSS) failure data are presented and discussed. We have also presented an assessment tool called R-ViSim, which is currently under development, to automate the simulation process. Practically, developers and project managers can use the proposed simulation procedure to estimate workloads that are similar to the actual situation and allocate appropriate human resources.

The remainder of this paper is organized as follows. In Section II, we give a brief review of model-based, queueing-based, and simulation-based software reliability analysis. In Section III, we will propose two queueing-based simulation procedures described in C-like programming language. The components in the queueing-based simulation procedures will be explained and discussed in detail. In Section IV, we apply three real data sets collected from OSS and CSS to discuss and assess the processes of fault detection and removal, and analyze the experimental results. We also address the threats to validity and answer some research questions in Section IV. Finally, conclusions and future research recommendations are offered in Section V.

II. RELATED WORKS

A. MODEL-BASED SOFTWARE RELIABILITY ANALYSIS

In the past, a lot of SRGMs were developed to estimate and predict the reliability of developed software systems [3], [4], [5], [6]. Some models assume that there is a finite and fixed number of faults in system while others assume that an infinite number [20]. Some models need the exact time in between each failure found in testing [21], while others only require the number of failures found during any given time interval [22] (i.e., a day or a week). Some SRGMs have been commonly used in the field of software reliability modeling. We will provide a brief review in the following.

1) THE GOEL AND OKUMOTO (G-O) MODEL

Goel and Okumoto [23] once proposed an exponential-type model, which describes a software detection phenomenon. The *Mean Value Function* (MVF) of the GO model is given by

$$m(t) = a(1 - e^{-bt}), \quad b > 0, \quad (1)$$

where a is the expected number of faults that will be eventually detected, and b is the faults detection rate per fault at an arbitrary testing time.

2) THE DUANE MODEL

This mode (also known as the *Weibull process model*) assumed that the MVF satisfies [22]

$$m(t) = \left(\frac{t}{k}\right)^b, \quad k > 0, b > 0, \quad (2)$$

where k and b are constant parameters which can be estimated using collected failure data.

3) THE YAMADA DELAYED S-SHAPED (DSS) MODEL

Yamada et al. [24] proposed a delayed S-shaped SRGM to describe the fault detection process. The observed growth curve of the cumulative number of detected faults is S-shaped. The MVF of the DSS model is given by

$$m(t) = a \left[1 - (1 + bt) e^{-bt} \right], \quad b > 0. \quad (3)$$

It is a two-parameter S-shaped curve with parameter a denoting the number of faults to be detected and b corresponding to the failure detection rate (and the fault isolation rate).

4) THE INFLECTED S-SHAPED (ISS) MODEL

This model was proposed by Ohba [25] and it described a software failure detection phenomenon with a mutual dependence of detected faults. The MVF of the ISS model is

$$m(t) = \frac{a(1 - e^{-bt})}{1 + \psi e^{-bt}}, \quad b > 0, \psi > 0, \quad (4)$$

where a is again the total number of faults to be detected, while b and ψ are the failure detection rate and the inflection factor, respectively.

5) THE LOGISTIC AND GOMPertz GROWTH CURVE MODELS

The Logistic and Gompertz growth curve models were also commonly used to estimate the fault content of developed software [26]. The expected cumulative number of detected faults at time t for the logistic growth curve model [27] can be described by

$$m(t) = \frac{a}{1 + ke^{-bt}}, \quad a > 0, b > 0, k > 0. \quad (5)$$

The MVF of the Gompertz growth curve model [27] is given by

$$m(t) = ak^{bt}, \quad a > 0, b < 1, k > 0, \quad (6)$$

where b and k are constant parameters which can be estimated by fitting the failure data. Notice that the parameter a in the above-mentioned models may typically be interpreted as the expected number of initial faults in the software.

6) TWO-ERROR-TYPES MODEL

Yamada et al. [28] proposed a modified exponential SRGM which assumes that the software system contains two types of faults, type I (which are easy to detect) and type II (which are difficult to detect). It is assumed that the faults detected early in testing are different from those detected later on. This model has a mean value function as follows:

$$m(t) = \sum_{i=1}^2 ap_i (1 - e^{-b_i t}), 0 < b_2 < b_1 < 1, \quad (7)$$

where p_i for ($i = 1, 2$), denoted the proportion of type i errors in the software system ($p_1 + p_2 = 1$), a is the expected number of faults and b_i is the error detection rate of type i errors per error (per unit time).

Debugging is not an easy thing. Most research has assumed that when faults are detected during testing, they will be removed immediately and successfully in debugging. In reality, the time required to remove the faults cannot be ignored [11], [29], [30], [31]. Kapur and Grag proposed a SRGM model for describing error removal [30]. In this SRGM, it is assumed that for a failure the detection of the faults causing the failure also results in the detection of a proportion of the remaining faults, without those faults causing any failure. Let $m_r(t)$ be the number of faults detected by time t . We then obtain

$$m_r(t) = \frac{ab(e^{(b+d)t} - 1)}{d + be^{(b+d)t}}, \quad (8)$$

where a is the total fault content of the software, b is the failure rate, and d is the error-detection rate of additional detected errors.

It is noted that Wu and Huang [10] once demonstrated how to derive mathematical expressions from the computational methods of deep learning models and how to determine the correlation between them and the mathematical formula of SRGMs. They used the back-propagation algorithm to obtain the SRGM parameters. However, the foregoing discussion showed that most studies assume that software reliability growth behavior follows a non-homogeneous poisson process (NHPP), based on collected software failure data. It is worth noting that Hou [33] reported that before applying the failure data to the SRGMs, whether the dataset is of the NHPP type and whether it is used correctly by the NHPP-based SRGM should be explained. On the other hand, Kanoun and Lapire [4] also observed that the use of NHPP-based SRGMs during the early stages of development and validation is much less convincing.

Additionally, Liu and Kang [34] once deduced an imperfect debugging software belief reliability growth model using the uncertain differential equation under the framework of

uncertainty theory, and investigated the properties of essential software belief reliability metrics. Garg et al. [35] proposed an entropy-combinative distance-based assessment (CODAS-E) method and presented to select and rank SRGMs based on multiple performance indexes. Nafreen and Fiondella [36] proposed a framework for SRGM possessing a bathtub-shaped fault detection rate and derived stable and efficient expectation conditional maximization algorithms to fit the models. Sabnis and Joshi [37] ever proposed an architecture to enhance, optimize and validate software reliability using machine learning techniques. Yang et al. [38] proposed a quantitative reliability evaluation process and method of network system based on discrete event simulation combining software and hardware.

B. QUEUEING-BASED SOFTWARE RELIABILITY MODELING

It is worth noting that most of the traditional SRGMs are usually based on the same assumptions. Yet some of the assumptions are unreasonable, particularly regarding the development of OSS, such as with perfect debugging and immediate debugging. For example, in practice, the assumption about perfect debugging may not be reasonable. This assumption states that when a failure is detected, developers correctly remove the corresponding fault without introducing new faults.

In reality, developers generally have the experience that as they remove a fault, they at times create other new faults. The assumption which hypothesizes perfect debugging is not legitimate, because the debugging process is a complex activity which includes locating and removing the relevant faults [39]. Therefore, the more complex the fault that developers want to remove is, the more complex the debugging process to be performed will be. For example, Raymond [40] once reported that beta testing plays a key role in the testing of OSS, hence the test team usually is not the same as the development team in OSS. Thus, once a fault is detected, developers of OSS usually need more time to communicate with testers for removing the fault. The above situation makes debugging time lag occur more easily in OSS.

In the past, some researchers once proposed to use *Infinite Server Queueing* (ISQ) and/or *Finite Server Queueing* (FSQ) approaches to model the FDP and the FCP [9], [11], [14], [15]. Basically, both ISQ and FSQ approaches are based on queuing theory to make projections on software reliability. In general, a queueing system can be described as customers arriving for service, waiting for service if it is not immediate, and leaving the system after being served [41]. We observed that some characteristics of this theory are similar to those of software engineering:

- *Arrive rate of customers*: The concept is similar to the fault detection rate of the software system. If the probability distribution of the arriving process is time-independent (the arrival rate does not change with time),

it is defined as stationary. If the probability distribution is not time-independent, it is *non-stationary*.

- *Service time*: Similar to debugging time, which is the average time taken for debugging by a debugger in the queueing system. Similar to arrivals, it can be stationary or non-stationary.
- *Number of service channels*: The service channels may be viewed as the debuggers who can analyze and process the detected faults concurrently.
- *Queue discipline*: The rule by which faults are selected for debugging when a waiting queue has formed. The rule can be First-Come-First-Served (FCFS), Last-Come-First-Served (LCFS), random, or a priority scheme.

The standard shorthand for describing the queueing system is Kendall. It consists of a series of symbols and slashes such as $A/B/X/Y/Z$, where A represents the arrival time distribution, B represents the probability distribution of the service time, X represents the number of total servers in system, Y is the limits of the system capacity, and Z is the rule for queue discipline. A and B in shorthand can be an exponential distribution (M), a deterministic distribution (D), a type- k Erlang distribution (E_k), a phase type distribution (PH), or a general distribution (G). The service rule Z can be FCFS, LCFS, random selection for service (RSS), priority (PR), or general discipline (GD). Y and Z are usually omitted if there are no constraints on the queue length and the queue discipline is FCFS [41].

In the past, Lin et al. [9] once proposed a preemptive priority queueing (PPQ) model that considers both a finite number of debuggers and different priority levels. Inoue and Yamada [15] also proposed an ISQ model considering the time distribution of the fault-isolation process based on DSS models. This model can easily describe and analyze fault detection during the actual testing phase and expresses several NHPP-type models as special cases. Gokhale and Mullen [42] have presented the multi-priority queueing models to estimate the mean resolution time resolution of faults with different severity levels. They found that the priority model mixed non-preemptive priority and preemptive priority principle is suitable to describe the defect data. Kapur et al. [43], [44] and Dohi et al. [45] also discussed how to use the ISQ model to describe software development behavior.

Furthermore, Zhang [46] incorporated detection and correction efforts into the fault detection and correction processes, respectively. Huang et al. [47] proposed an extended ISQ model with multiple change-points to estimate the software reliability. Zhang et al. [48] showed how to apply ISQ models with testing effort functions (TEFs) to model the FDP and FCP. Their proposed model with an Exponentiated Weibull TEF and a Logistic TEF can consider the influence of resources on the software debugging phase, and enhance the assessment of reliability.

Note that most studies assume that the detected faults are corrected on a FCFS basis [13], [15], [34], [44], [48].

However, Mockus et al. [16] found that in Apache and Mozilla, bugs with higher priority are fixed faster than bugs with lower priority. Zhang et al. [17] analyzed an IT management product for enterprise customer and collected its failure data. They calculated the time needed to fix the bugs and found that bugs with higher priority are fixed faster (as indicated by mean values). In the real world, the debugging team to fix the faults based on their severity and priority.

C. SIMULATION-BASED METHODS

Over the last few decades, simulation has proven to be an effective approach for analyzing software systems. It can also be used to address a variety of issues, such as management of software development, improving software processes, or training software project management. Simulation can provide an alternative method for investigating software reliability because it can model a wider range of reliability phenomena than mathematical analysis [4]. Kellner's study [49] could be the first state-based simulation applied to software engineering processes. Later on, Raffo and Kellner [50] discussed some of the important empirical issues that arise in software process simulation modeling, and proposed the software process simulation model and compared the simulated results to the real-world data. Raffo et al. [51] once addressed some relevant aspects of the multifaceted relationship between empirical studies and the building, deployment and usage of software process models.

Gokhale et al. [52] developed simulation procedures which may be used to assess the impact of individual components on the reliability of an application in the different repair strategies during testing. It can provide project owners with a range of operational configurations to meet the desired reliability. Rus et al. [53] described a process simulator to assist software project planning in the software development program. Using this discrete event model to construct a software reliability prediction model for a realistic project. Lin and Huang [54] once proposed simulation procedures based on queueing theory to describe and explain possible debugging behavior in the software development process. The proposed methods can help project leaders to effectively assess the appropriate staffing level for the debugging team. Juan et al. [55] developed a Java-based simulation software, J-SAEDES, which can estimate the reliability and availability of time-dependent computer systems and networks, and identify those components which play a critical role in system reliability and availability.

Additionally, Gokhale et al. [56] proposed a rate-based simulation framework that incorporates explicit debugging activities into SRGMs conducted according to different debugging policies. Their simulation framework obtains realistic software reliability estimates and determines the optimal software release time. Antoniol et al. [57] incorporated the queue theory and stochastic simulation to explore a real system and to evaluate its cost, risk, and staffing levels. Fan et al. [58] constructed a defect removal process

simulation based on finite independent queues with different capacities and loadings. The simulation takes into account the limitations on developers and the differing abilities of developers. It can also provide useful and important information, such as the duration of defect removal, the use of each developer in the defect removal process, and the rate of removed defects at a planned time.

Chang et al. [59] applied the express-queueing theoretic approach to model the fault correction process. All detected faults will be classified and dispatched into either an express queue or regular queue to reduce the mean resolution time. Their experiments show that the performance and efficiency of software debugging processes can be improved. Lin and Li [60] developed a new rate-based queueing simulation framework for open source software (OSS) reliability assessment. It can support the optimal release time decision and assist the OSS and CSS project managers to estimate the number of devoted core contributors. Shu et al. [61] proposed a simulation-based approach to easily analyze the reliability of fault tolerant web services and the execution details of different fault tolerant strategy. Thus, the developers can determine an appropriate strategy for different constraints. Nakahara et al. [62] also proposed a simulation model of software quality assurance to quantitatively demonstrate the positive effect of adding quality assurance effort especially in early phases of software development. Note that their proposed model can represent the relationship among the number of bugs in each phase, the amount of quality assurance effort, the expected number of detectable bugs and the amount of bug fixing effort.

III. SIMULATION-BASED FRAMEWORK FOR SOFTWARE RELIABILITY ESTIMATION

From above-mentioned studies, we can see that there is no single model or method that can be universally applicable to all the situations [4]. Here we will show an alternative approach to modeling the software reliability. In addition to our proposed queueing-based model [9], we find that simulation procedures can also be used to investigate the software fault correction process and to estimate system performance measures such as the average debugging time and waiting time. Specifically speaking, the queueing-based simulation approaches can relax certain unreasonable assumptions which are common in model-based approaches. In order to mimic the actual situation of FCP, we incorporate the concept of preemptive priority queueing into the rate-based simulation procedure. In this section, the details of the debugging behavior are discussed and analyzed using the fault priority level.

The assumptions of the queueing-based model with preemptive priority queueing (PPQ) policy are as follows [9], [63], [64], [65], and [66]:

- 1) The software is subject to failures at random times caused by the manifestation of remaining faults in the system.

- 2) The mean number of detected faults in the time interval $(t, t + \Delta t)$ is proportional to the mean number of remaining faults in the system.
- 3) The detected faults will be put into the waiting queue according to a Poisson process with detection rate (λ) . Additionally, the detected faults are fixed by a finite number of debuggers (c) ; The fault correction time for each assigned debugger is exponentially distributed with correction rate (μ) .
- 4) There are two different types of faults: *high-priority* faults and *low-priority* faults. High-priority faults have a preemptive ability for service over low-priority faults. If the two faults have the same priority, they will be served according to their order in the waiting queue.
- 5) If all debuggers are busy with high-priority faults, a newly detected high-priority fault follows the FCFS rule and waits at the tail of the same priority faults in the waiting queue.
- 6) The waiting time and correction time of fault(s) are mutually independent. Fault detection activity continues while faults are removed, and fault correction does not affect the detection process.
- 7) Fault correction time is non-negligible. When a fault is fixed, it will not introduce a new fault.

Based on the above assumptions, we will be able to obtain a queueing model that describes the fault removal process, $M/M/c/\infty/PR$. Figure 1 shows the rate transition diagram of proposed queueing-based model with PPQ policy. To implement the procedures, a C-like programming language will be used, but there are many possibilities. Figure 2 shows the flowchart of our proposed queueing-based simulations. Note that each iteration in the procedure includes three steps. Note that the staff allocation, the first step, assigns a fault to a free debugger. The second step is used to determine whether a fault is detected according to the failure data set. The end of an iteration is the step of fault correction, which checks whether the faults are completely corrected. The three steps will be repeated until the time for the simulation procedure runs out of the total execution time.

A. PROCEDURE #1: THE SIMULATION PROCEDURE FOR THE NON-PREEMPTIVE PRIORITY SYSTEM

The flowchart of the Procedure #1 is shown in Figure 3. Additionally, the algorithm shown in Figure 4 presents the non-preemptive debugging simulation procedure. Under non-preemptive scheduling, once a process goes into operation, it will not be interrupted until completion. For example, developers often practically deal with higher-priority faults more rapidly than lower-priority faults. The Procedure #1 accepts three parameters as inputs: the total execution time, defined as *stop_time*, the length of time for each iteration, *dt*, and *staffing_level*, the total number of debuggers in the system. Further, we split the execution time into a great number of iterations. Thus, the probability of multiple events occurring within each iteration is negligible [4], [7], [52].

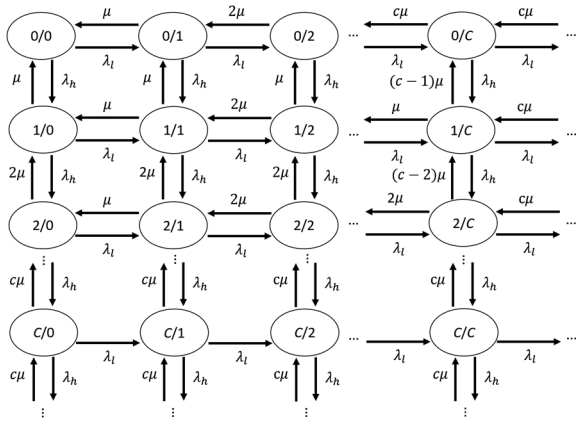


FIGURE 1. Rate transition diagram of M/M/c queue with two priority classes.

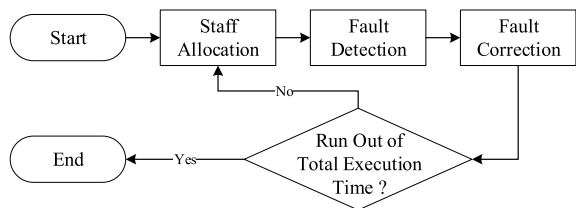


FIGURE 2. The flowchart of queueing-based simulation procedure.

Note that *current_time* is a variable representing the current cumulative simulation execution time. After the completion of each iteration, the time spent on iteration (*dt*) will be added to the current execution time until it reaches the total execution time (*stop_time*). As the current execution time increases, our system updates the state of faults (processing or waiting) to record the time cost in each component. We denote the number of busy debuggers at present by *working_server*. The *Set* data structure *correcting_set* is used to collect faults being processed. Furthermore, the *high_priority_waiting_queue* and *low_priority_waiting_queue* are the *Queue* data structure used to store the waiting high priority faults and low priority faults, respectively. Based on the assumptions described in this section, Procedure #1 was developed and implemented in three phases: staff allocation, fault detection, and fault correction.

1) PART 1 STAFF ALLOCATION PHASE

In this phase, we firstly check whether there are available debuggers in the system by comparing the number of busy debuggers (*working_server*) with all debuggers (*staffing_level*). If unoccupied debuggers actually exist, we check whether there are faults in the waiting queues according to their priority orders (high priority faults first). For example, if there is a waiting high priority fault, the fault will be assign to a debugger and its status will change from waiting to processing. If the number of waiting high priority faults is zero, then we check whether there are low priority faults in the queue. In short, the high priority waiting queue takes

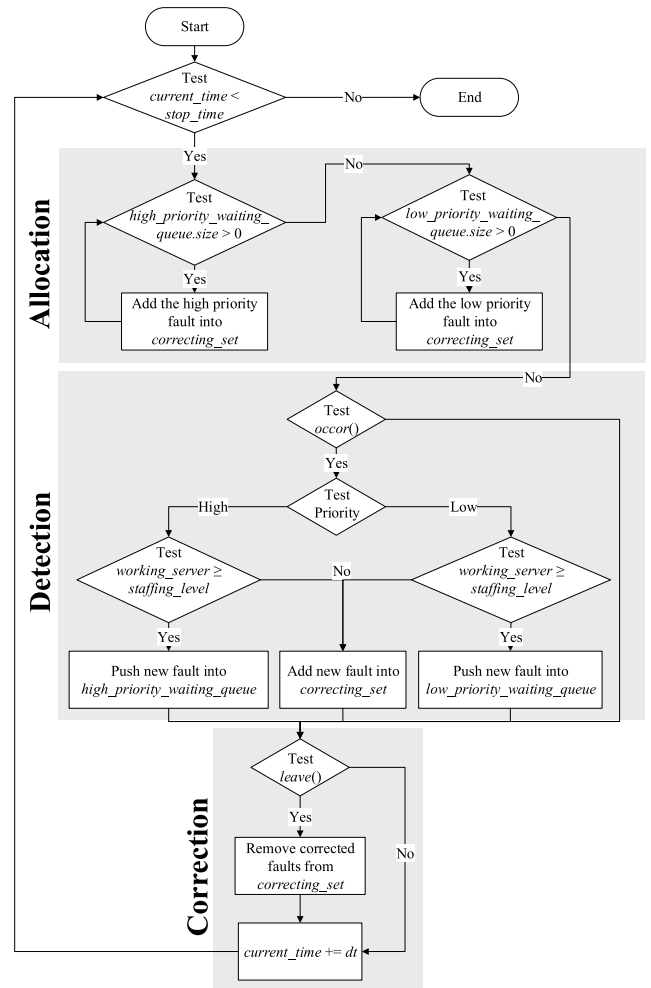


FIGURE 3. The flowchart of non-preemptive debugging simulation procedure.

precedence over the low priority waiting queue. These actions are described in lines 9-18 of Figure 4, which will repeat until there are no waiting faults or there are no available debuggers.

2) PART 2 FAULT DETECTION PHASE

Note that the detection time of faults is based on the real failure dataset we collected from OSS and CSS projects. Each fault will enter the simulation system based on timestamp ordering (time detected). Therefore, in each iteration, the function *occur()* compares the timestamps with the current execution time (*current_time*) to check whether any fault is detected. When the *occur()* function returns true, the priority of the detected fault will be determined and handled independently. For example, if a low priority fault is detected, it will be added to the low priority waiting queue, and assigned to a debugger if there are available debuggers. These activities are shown in lines 20-37 of Figure 4.

3) PART 3 FAULT CORRECTION PHASE

At the beginning of this step, each fault being processed will be checked by the *leave()* function which is used to determine

```

01 void Simulation_Procedure(double stop_time, double dt, int
      staffing_level)
02 {
03     double current_time = 0 ;
04     int working_server = 0 ;
05     set correcting_set = {} ;
06     Queue high_priority_waiting_queue,
      low_priority_waiting_queue ;
07     while( current_time < stop_time ) {
08         // Staff Allocation:
09         while ( working_server < staffing_level ) {
10             if ( high_priority_waiting_queue.size > 0 ) {
11                 working_server++ ;
12                 correcting_set ← correcting_set U
      high_priority_waiting_queue.dequeue() ;
13             }
14             else if( low_priority_waiting_queue.size > 0 ) {
15                 working_server++ ;
16                 correcting_set ← correcting_set U
      low_priority_waiting_queue.dequeue() ;
17             }
18         }
19         // Fault Detection:
20         if ( occur() ) {
21             if ( fault ∈ HIGH_PRIORITY ) {
22                 if ( working_server ≥ staffing_level )
23                     high_priority_waiting_queue.enqueue(fault) ;
24                 else {
25                     working_server++ ;
26                     correcting_set ← correcting_set U fault ;
27                 }
28             }
29             else if ( fault ∈ LOW_PRIORITY ) {
30                 if ( working_server ≥ staffing_level )
31                     low_priority_waiting_queue.enqueue(fault) ;
32                 else {
33                     working_server++ ;
34                     correcting_set ← correcting_set U fault ;
35                 }
36             }
37         }
38         // Fault Correction:
39         for ( ∀ fault ∈ correcting_set ) {
40             if ( leave(fault) ) {
41                 working_server-- ;
42                 correcting_set ← correcting_set - fault ;
43             }
44         }
45         current_time += dt;
46     }
47 }

```

FIGURE 4. Non-preemptive debugging simulation procedure.

whether the fault will be corrected. The probability that a debugger successfully corrects the fault in the time interval $(t_s, t_s + \Delta t)$, given that it has already been in progress for time t_s is [54]:

$$P(t_s \leq T_s \leq t_s + \Delta t | T_s > t_s) = \mu \times \Delta t, \tag{9}$$

where μ represents the fault correction rate, and T_s represents the time spent on fault correction. Thus, we can generate a random number x which is the probability of correcting a fault in a unit time and compared it with $\mu \times \Delta t$ after invoking the *leave()* function. When x is greater than $\mu \times \Delta t$, the devoted debugger will successfully correct the fault in this

iteration. Otherwise, a fault which cannot be corrected will be rechecked in the next iteration. These actions are given in lines 39-44 of Figure 4.

B. PROCEDURE #2: THE SIMULATION PROCEDURE FOR THE PREEMPTIVE PRIORITY SYSTEM

Here we will further discuss and present the simulation procedure of the preemptive priority system in which high priority faults can preempt low priority faults. The flowchart of the simulation procedure is depicted in Figure 5 and the algorithm of this procedure is shown in Figure 6. Note that Procedure #2, constructed on the basis of Procedure #1, also accepts three parameters as inputs:

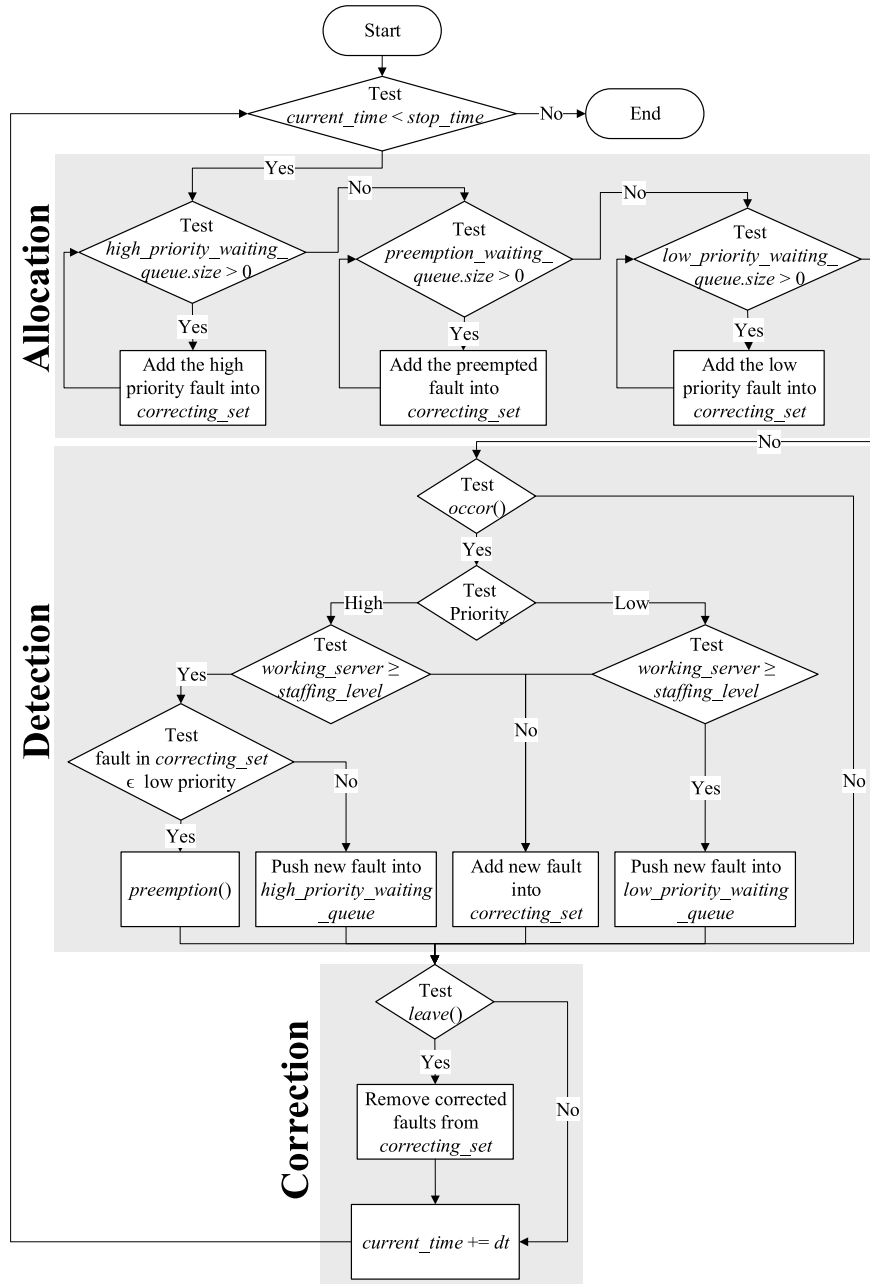


FIGURE 5. The flowchart of preemptive debugging simulation procedure.

current_time, *dt*, and *staffing_level*. Owing to the preemptive property of the Procedure #2, the Queue data structure, *preemption_waiting_queue*, is used to place the preempted faults (low priority faults).

Based on the assumptions described in this section, Procedure #2 was developed and implemented in three phases: staff allocation, fault detection, and fault correction.

1) PART 1 STAFF ALLOCATION PHASE

First, we check the available debuggers by comparing *working_server* with *staffing_level*. The activity is the same as the staff allocation of the Procedure #1. Based on our

assumptions in Section III, preempted faults are handled prior to waiting low priority faults and faults with high priority take priority over preempted faults. Therefore, the available debuggers will be allocated according to the order of priority: waiting high priority faults, preempted faults, and waiting low priority faults. The details of actions for staff allocation are described in lines 10-23 of Figure 6, which is repeated until there are no waiting faults or no available debuggers.

2) PART 2 FAULT DETECTION PHASE

When a fault is detected, we then check what kind of priority it has. For instance, if a high priority fault enters


```

01 void Simulation_Procedure(double stop_time, double dt, int
    staffing_level)
02 {
03     double current_time = 0 ;
04     int working_server = 0 ;
05     set correcting_set = {} ;
06     Queue high_priority_waiting_queue, preemption_waiting_queue,
07         low_priority_waiting_queue ;
08     while( current_time < stop_time ) {
09         // Staff Allocation:
10         while ( working_server < staffing_level ) {
11             if ( high_priority_waiting_queue.size > 0 ) {
12                 working_server++ ;
13                 correcting_set ← correcting_set U
14                     high_priority_waiting_queue.dequeue() ;
15             }
16             else if( preemption_waiting_queue.size > 0 ) {
17                 working_server++ ;
18                 correcting_set ← correcting_set U
19                     preemption_waiting_queue.dequeue() ;
20             }
21             else if( low_priority_waiting_queue.size > 0 ) {
22                 working_server++ ;
23                 correcting_set ← correcting_set U
24                     low_priority_waiting_queue.dequeue() ;
25             }
26         }
27         // Fault Detection:
28         if ( occur() ) {
29             if ( fault ∈ HIGH_PRIORITY ) {
30                 if ( working_server ≥ staffing_level ) {
31                     ( ∃ {fault | fault ∈ LOW_PRIORITY } ⊆
32                         correcting_set ) ? preemption(fault) :
33                         high_priority_waiting_queue.enqueue(fault) ;
34                 }
35                 else {
36                     working_server++ ;
37                     correcting_set ← correcting_set U fault ;
38                 }
39             }
40             else if ( fault ∈ LOW_PRIORITY ) {
41                 if ( working_server ≥ staffing_level )
42                     low_priority_waiting_queue.enqueue(fault)
43                 else {
44                     working_server++ ;
45                     correcting_set ← correcting_set U fault ;
46                 }
47             }
48         }
49         // Fault Correction:
50         for ( ∀ fault ∈ correcting_set ) {
51             if ( leave(fault) ) {
52                 working_server-- ;
53                 correcting_set ← correcting_set - fault ;
54             }
55         }
56         current_time += dt;
57     }
58 }

```

FIGURE 6. Preemptive debugging simulation procedure.

the system and no debuggers are available in this iteration, we check whether there is a low priority fault being processed in the *correcting_set*. If there are low priority faults which are being processed, the *preemption()* function will replace the first entering low priority fault in the *correcting_set* with the high priority one, and the

preempted fault will be sent to the *preemption_waiting_queue*. Otherwise, the newly detected high priority fault will be placed in the *high_priority_waiting_queue*. Details of the preemption procedure are shown in Figure 7 and the fault detection part is described in lines 25-44 of Figure 6.

```

01 void preemption (high_priority_fault)
02 {
03     for ( ∀ fault ∈ correcting_set ) {
04         if ( fault ∈ LOW_PRIORITY ) {
05             correcting_set ← correcting_set - fault ;
06             preemption_waiting_queue.enqueue(fault) ;
07             break ;
08         }
09     }
10     correcting_set ← correcting_set U high_priority_fault ;
11 }

```

FIGURE 7. Preemption function.

3) PART 3 FAULT CORRECTION PHASE

In each iteration, the *correcting_set* will be checked, and the faults that are successfully corrected by debuggers will be removed from *correcting_set* and the number of busy debuggers will decrease by one. When the currently processed faults are corrected, the occupied debuggers will be available at the same time. These actions are described in lines 46-51 in Figure 6. If a fault cannot be corrected successfully in the current iteration, it will be checked in the next iteration.

IV. NUMERICAL EXAMPLES

A. SELECTED DATA SETS AND MODEL'S COMPARISON CRITERIA

In this paper, the failure data used for the evaluations in this paper are composed of three sets of data and come from three sources [9], [67], [68]. The first data set (DS1) was collected from the public bug-tracking system, Bugzilla [67]. The system contains the shared components used by Firefox and other Mozilla software. The second data set (DS2) was also obtained from Bugzilla [68]. The third data set (DS3) was collected from the Coretronic Corp. It is noted that in software reliability engineering, the overall failure data sets typically fall into two types: time domain data and interval domain data [2], [3]. Practically, the time domain data provides better accuracy in the parameter estimates, but it also inevitably involves more data collection efforts and computations than the interval domain approach [2], [3], [4]. In our experiments, these three data sets were collected in interval domain format. Table 1 displays a sample of the actual failure data. Additionally, Table 2 shows the data source and system characteristics for each set of failure data.

On the other hand, Figure 8 depicts the cumulative number of detected and corrected high/low priority faults versus the time of the three data sets. The difference between detected and removed faults are the open-remaining (detected but in the waiting queue) faults. They clearly show that the fault removal time is not negligible because the total number of removed faults obviously lags behind the total number of detected faults.

In this paper, except for our proposed simulation-based method, five selected SRGMs are selected for performance comparisons. They are the GO model [4], the DSS model [4], the ISS model [4], the Gompertz model [8], and the PPQ model [9]. The method of maximum likelihood estimation is used to estimate the parameters of all models [2], [3], [4], [8]. The following criteria are used to evaluate 11 selected models.

- 1) The *Mean Square Error* (MSE) is defined by [4], [8], and [9]:

$$MSE = \frac{1}{n - \theta} \sum_{i=1}^n (m_i - m(t_i))^2, \quad (10)$$

where m_i is the cumulative number of detected faults in a given time interval $(0, t_i]$, $m(t_i)$ is the mean value function, i.e., the expected number of software failures by time t_i , n is the size of the selected data set, and θ is the degree of freedom. The lesser the MSE, the better the model performance.

- 2) The *Theil's U Statistic* (TS) includes two parts, denoted $U1$ and $U2$. They are defined as follows [69], [70]:

$$U1 = \frac{\sqrt{\sum_{i=1}^n (m_i - m(t_i))^2}}{\sqrt{\sum_{i=1}^n m_i^2 + \sum_{i=1}^n m(t_i)^2}}, \quad (11)$$

and

$$U2 = \frac{\sqrt{\sum_{i=1}^{n-1} \left(\frac{m(t_{i+1}) - m_i}{m_i} \right)^2}}{\sqrt{\sum_{i=1}^{n-1} \left(\frac{m_{i+1} - m_i}{m_i} \right)^2}}, \quad (12)$$

A low value of $U1$ and $U2$ means that the method provided a more accurate prediction.

- 3) The *Coefficient of Determination* (R^2) is defined as [4]:

$$R^2 = 1 - \frac{\sum_{i=1}^n (m(t_i) - m_i)^2}{\sum_{i=1}^n (m_i - \bar{m})^2}, \quad (13)$$

and

$$\bar{m} = \frac{\sum_{i=1}^n m_i}{n}, \quad (14)$$

A higher R^2 value indicates the model is a good fit.

B. DS1

Here we will apply the PPQ simulation procedure described in Section III to DS1. The simulation procedures were implemented with Python and the experiments executed on an AMD FX-6350 machine with six 3.9 GHz cores and 8GB of memory, running in a Windows 10 environment. Note that there are two steps in our proposed simulation procedure algorithm. The first step is to calculate the parameters from the data set. The second step is to conduct the simulation and average the experimental results. Using DS1, we simulate the proposed PPQ model for a period of 45 weeks and set each time unit iteration as 0.001 week. Based on our past study in [9], first, Table 3 shows a summary of the DS1 parameters used in the simulation procedure for DS1. Note that c , λ_h , λ_l , and μ are the parameters that stand for the number of debuggers, the high-priority fault detection rate, the low-priority fault detection rate, and the fault correction rate, respectively [9]. From DS1, we obtain the average detection rates of high and low priority faults, 1.244 ($= 56/45$) and 0.555 ($= 25/45$) faults per week, respectively. To ensure

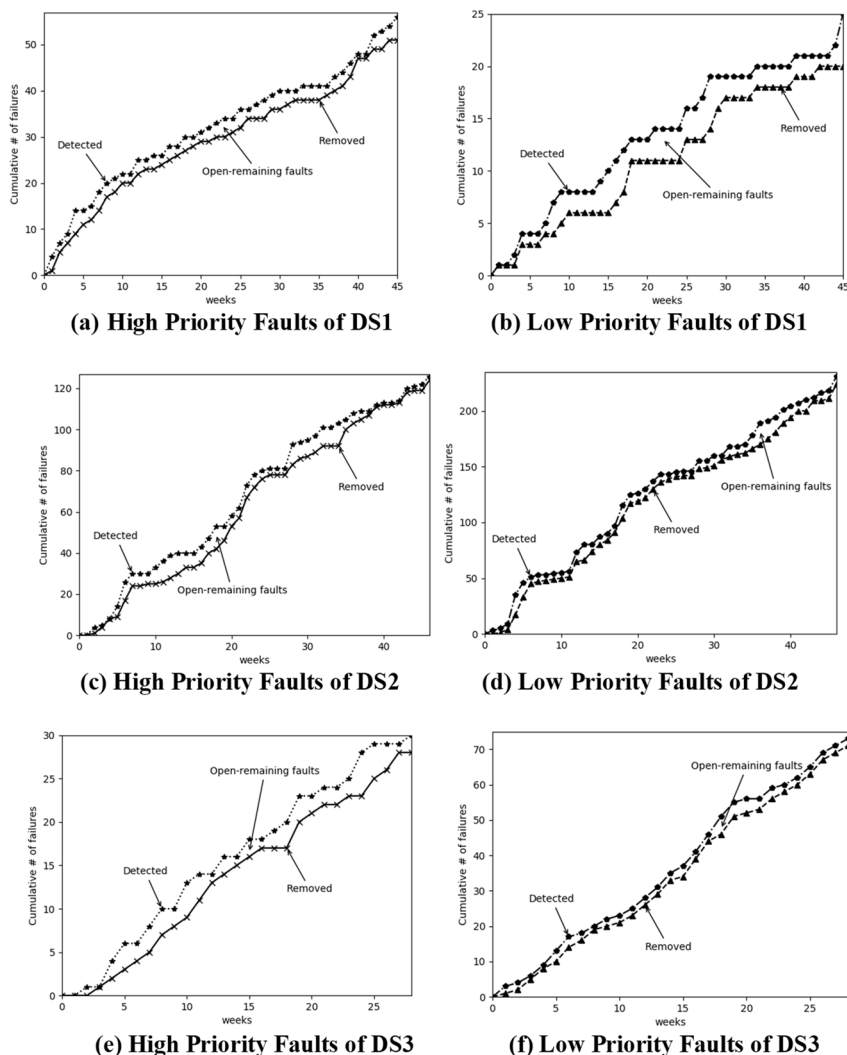


FIGURE 8. Cumulative number of detected and corrected faults for DS1-DS3.

TABLE 1. A sample of the actual failure data.

Information on each fault				Cumulated number of faults		
Fault ID	Priority	Arrival time	Fixed time	Time (weeks)	# of high priority faults detected	# of low priority faults detected
1	Low	2013/1/2 03:11	2013/2/18 13:44	1	4	1
2	High	2013/1/2 13:15	2013/1/4 10:13	2	7	1
3	Low	2013/1/3 04:03	2013/3/6 02:43	3	9	2
4	High	2013/1/3 11:21	2013/1/5 16:35	4	14	4
⋮				⋮		

TABLE 2. Characteristics of failure data sets.

Selected Data Sets	Duration	No. of Failures	Remarks
DS1 [67]	311 Days	81	The data collection period was from Jan. 2, 2013 to Nov. 9, 2013.
DS2 [68]	318 Days	357	The data collection period was from Jan. 2, 2002 to Nov. 16, 2002.
DS3 [9]	192 Days	103	The data collection period was from Feb. 4, 2014 to Aug. 15, 2014.

system stability ($\rho = (\lambda/c\mu) < 1$), we assume that the service rate for each debugger μ is 0.201 faults per week, and

the number of debuggers c is 10. Table 3 gives a summary of the DS1 parameters used in the simulation procedure for

TABLE 3. Parameters used in the simulation procedure of DS1.

Parameters	c (person)	λ_h (fault/week)	λ_l (fault/week)	μ (fault/week)	dt (week)	$stop_time$ (week)
Value	10	1.244	0.555	0.201	0.001	45

TABLE 4. Performance evaluation and comparison of different models and methods for corrected faults of DS1.

Models (High Priority)	MSE	R ²	TS-U1	TS-U2
GO model [9]	13.421	0.920	0.053	0.847
DSS model [9]	25.217	0.850	0.074	1.185
ISS model [9]	12.136	0.924	0.052	0.927
Gompertz model [9]	17.325	0.897	0.060	1.086
PPQ model [9]	8.453	0.949	0.043	0.678
Simulation-Based Method	5.152	0.969	0.037	0.533
Models (Low Priority)	MSE	R ²	TS-U1	TS-U2
GO model [9]	1.217	0.970	0.042	0.716
DSS model [9]	1.744	0.971	0.049	0.845
ISS model [9]	1.201	0.813	0.094	1.061
Gompertz model [9]	1.345	0.967	0.043	1.070
PPQ model [9]	1.077	0.973	0.039	0.636
Simulation-Based Method	1.174	0.969	0.043	0.629

DS1. We repeat the simulation procedure for 1,000 times and calculate the average of the experimental results.

Table 4 summarizes the performance comparisons of selected models for corrected faults of DS1. The proposed simulation-based method is the best for high priority faults of DS1 in terms of MSE, R², TS-U1, and TS-U2. As for the low priority faults of DS1, the proposed simulation-based method almost performs better than the traditional SRGMs. The PPQ model only shows a significantly better performance than the simulation-based method based on the values of MSE, R², and TS-U1. In general, we can see that the proposed simulation-based method almost gives the lowest values for TS-U1 and TS-U2 compared to traditional SRGMs for DS1. Obviously, we can see from Table 4 that both the proposed simulation-based method and PPQ model provide a better fit for DS1 and predict future fault correction processes well. Since real life debugging teams deal with high priority faults first, it can be seen that the preemptive mechanism better simulates the real world situation.

Figure 9 illustrates the cumulative corrected faults of the actual failure data DS1 versus the simulated failure data, which are generated by the procedures described in Section III-B. We can thus investigate the profiles of the actual failure data and the simulation results. We can see from Figure 9 that there is an obvious difference between the actual data and the simulation results in high priority faults between the 2nd and 12th weeks. The simulation results exhibit similar results for low priority faults between the 2nd and 13th weeks. This probably because we only use part of the actual data, ignoring the previous faults being processed and the open-remaining faults. Overall, the approximation of the curves illustrating the cumulative number of corrected faults are close to each other in Figure 9.

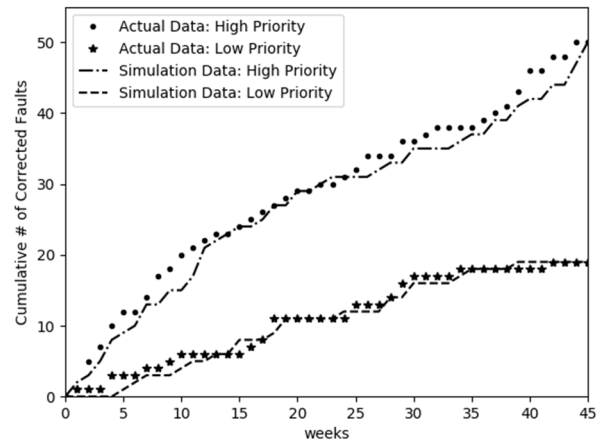


FIGURE 9. Plot of the actual and simulated failure data (DS1).

Finally, the comparison results of performance measurement between model-based and simulation-based methods are highlighted in Table 5. The simulation results and the performance estimation of the PPQ model are very close in average queue length, average waiting time, and the average response time, and the relative errors are within -0.14%. The results show that our proposed simulation procedure is workable to predict future behavior well.

C. DS2

Similarly, we repeat the simulation experiment steps based on the failure data of DS2. Thus, the average detected rate of high priority faults and low priority faults are 2.739 (=126/46) and 5.021 (=231/46), respectively. To ensure system stability ($\rho = (\lambda/c\mu) < 1$), we assume that the service rate for each debugger μ is 0.24 fault per week, and the number of debuggers c is 40. In Table 6, we show a summary of the parameters used in the simulation procedure for DS2.

Table 7 summarizes the performance comparisons of selected models for corrected faults of DS2. The proposed simulation-based method is the best for high- and low-priority faults of DS2 in terms of MSE, R², and TS-U1. As for the low priority faults of DS2, the proposed simulation-based method almost performs better than the traditional SRGMs. The PPQ model only shows a significantly better performance than the simulation-based method based on the values of MSE, R², and TS-U1. Although the ISS model has smaller values of TS-U2 compared to the proposed simulation-based method, but the proposed simulation-based method still shows a significantly better performance than other SRGMs and the PPQ

TABLE 5. Comparison results between modeling and simulation for DS1.

Performance Measures	Model-Based Method [9]	Simulation-Based Method	Relative Error
Avg. high priority waiting queue length	0.1938	0.1938	-0.02%
Avg. low priority waiting queue length	5.3830	5.3790	-0.07%
Avg. high priority waiting time	0.1558	0.1557	-0.06%
Avg. low priority waiting time	9.6690	9.6830	-0.14%
Avg. high priority response time	5.1309	5.1308	-0.00%
Avg. low priority response time	14.674	14.658	-0.10%

TABLE 6. Parameters used in the simulation procedure of DS2.

Parameters	c (person)	λ_h (fault/week)	λ_l (fault/week)	μ (fault/week)	dt (week)	$stop_time$ (week)
Value	40	2.739	5.021	0.24	0.001	46

TABLE 7. Performance evaluation and comparison of different models and methods for corrected faults of DS2.

Models (High Priority)	MSE	R ²	TS-U1	TS-U2
GO model [9]	34.048	0.974	0.039	1.368
DSS model [9]	31.560	0.982	0.036	0.754
ISS model [9]	35.123	0.976	0.041	0.481
Gompertz model [9]	43.315	0.971	0.045	1.692
PPQ model [9]	25.664	0.983	0.034	1.427
Simulation-Based Method	25.792	0.983	0.034	0.709
Models (Low Priority)	MSE	R ²	TS-U1	TS-U2
GO model [9]	94.247	0.977	0.034	2.936
DSS model [9]	178.62	0.957	0.047	4.646
ISS model [9]	90.751	0.978	0.034	3.182
Gompertz model [9]	178.37	0.957	0.047	4.099
PPQ model [9]	67.981	0.983	0.030	2.766
Simulation-Based Method	48.354	0.988	0.026	2.729

model based on the value of TS-U2 for high priority faults of DS2. Again, we can see from Table 7 that the proposed simulation-based method and PPQ model provide a better fit for DS2 and are generally more accurate than the traditional SRGMs.

In Figure 10, we plot the actual and simulated failure data for DS2. We can see from Figure 10 that the simulation results for the high priority faults and low priority faults have a small difference with the actual data and the simulation data in the period from the 4th week to the 10th week. This is probably because we ignore the previous faults being processed and the open-remaining faults. Overall, the approximation of the curves depicting the cumulative number of corrected faults are close to each other in Figure 10. Table 8 shows the comparison results between model-based and simulation-based methods for DS2. It can be found that the simulation results and measurements from the PPQ model are very close to each other in average queue length, the average waiting time, and the average response time, and the relative errors are within 0.08%. Note that for high priority faults the average queue length and average waiting time are zero. This is because the simulation results are too small to be measured. Thus, the relative error cannot be obtained. These experimental results indicate that our proposed simulation procedure is workable to predict faults in the correction process well for DS2.

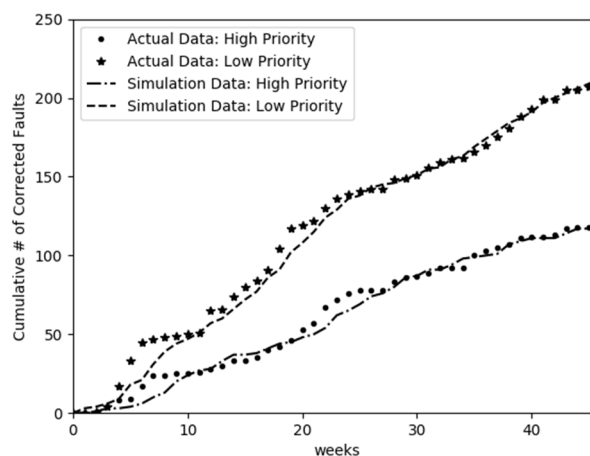


FIGURE 10. Plot of the actual and simulated failure data (DS2).

D. DS3

Similarly, we repeat the experiment steps based on the failure data of DS3. Then, the average fault detection rate of high and low priority faults is 1.071(=30/28) and 2.607(=73/28) fault per week, respectively. To ensure system stability ($\rho = (\lambda/c\mu) < 1$), we assume that the fault correction rate for each debugger μ is 0.55 faults per week and that the number of debuggers c is 7. In Table 9, we give a summary of the DS3 parameters used in the simulation procedure.

Table 10 summarizes the performance comparisons of selected models for corrected faults of DS3. The proposed simulation-based method is the best for low priority faults of DS3 in terms of MSE, R², and TS-U1. As for the high priority faults of DS3, the proposed simulation-based method almost performs better than the traditional SRGMs in terms of MSE and TS-U2. In general, both the proposed simulation-based method and PPQ model provide a better fit for DS3 and are generally more accurate than the traditional SRGMs.

The cumulative number of actual failure data in the DS3 dataset versus the simulated failure data is illustrated in Figure 11. The simulation result corresponds well to the actual failure data. This may be because there are very few

TABLE 8. Comparison results between modeling and simulation for DS2.

Performance Measures	Model-Based Method [9]	Simulation-Based Method	Relative Error
Avg. high priority waiting queue length	1.49×10^{-11}	0	-
Avg. low priority waiting queue length	0.5752	0.5755	0.05%
Avg. high priority waiting time	5.45×10^{-12}	0	-
Avg. low priority waiting time	0.1145	0.1146	0.08%
Avg. high priority response time	4.1667	4.1666	-0.00%
Avg. low priority response time	4.2812	4.2806	-0.01%

TABLE 9. Parameters used in the simulation procedure of DS3.

Parameters	c (person)	λ_h (fault/week)	λ_l (fault/week)	μ (fault/week)	dt (week)	$stop_time$ (week)
Value	7	1.071	2.607	0.550	0.001	28

TABLE 10. Performance evaluation and comparison of different models and methods for corrected faults of DS3.

Models (High Priority)	MSE	R ²	TS-U1	TS-U2
GO model [9]	1.054	0.980	0.037	1.629
DSS model [9]	1.105	0.974	0.044	0.973
ISS model [9]	1.322	0.962	0.053	1.043
Gompertz model [9]	1.475	0.963	0.050	1.348
PPQ model [9]	0.966	0.982	0.035	0.883
Simulation-Based Method	0.966	0.973	0.042	0.891
Models (Low Priority)	MSE	R ²	TS-U1	TS-U2
GO model [9]	10.05	0.980	0.036	1.527
DSS model [9]	8.423	0.985	0.033	1.225
ISS model [9]	7.356	0.985	0.032	0.932
Gompertz model [9]	13.83	0.973	0.042	1.532
PPQ model [9]	3.133	0.993	0.020	0.707
Simulation-Based Method	2.241	0.996	0.017	0.920

open-remaining faults in the past. In Table 11, we have highlighted the comparison results between model-based and simulation-based methods for DS3. From Table 11, we see that the simulation results and measurements from the PPQ model are very close to each other in average queue length, the average waiting time, and average response time, and the relative errors are within 0.11%. Note that the high priority fault has an average queue length and average waiting time of zero. This is because the simulation results are too small to measure. Thus, the relative error cannot be obtained. These experimental results show that the proposed simulation procedure predicts future fault correction processes well using the DS3 dataset.

E. RELIABILITY VISUALIZATION SIMULATOR

In order to visualize the simulation results, we have developed an assessment tool called the Reliability Visualization Simulator (R-ViSim) to provide quantitative estimation based on the PPQ model. R-ViSim’s major functions are listed below.

- **PPQ modeling.** R-ViSim can calculate the performance measures of the PPQ model [9] such as average queue length, average waiting time, and average response time and the parameter settings.

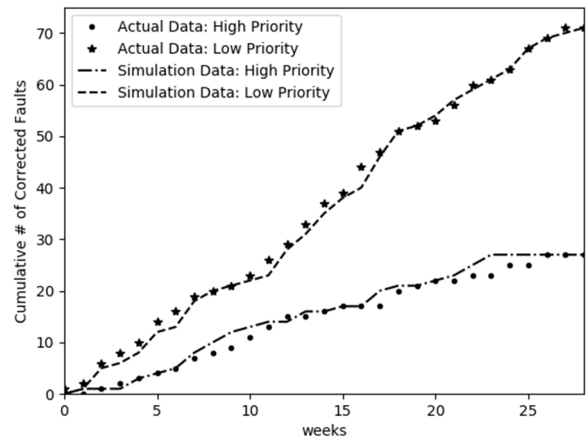


FIGURE 11. Plot of the actual and simulated failure data (DS3).

- **PPQ simulation.** Based on real failure data, R-ViSim can simulate the fault correction process using the preemptive priority queueing simulation procedure.
- **The results display.** R-ViSim can graphically display the cumulative corrected faults of the actual failure data versus the simulated failure data, which are generated by the PPQ simulation. The measurement derived from the PPQ modeling is filled in the blank of simulation results.

The high level architecture of developed R-ViSim is shown in Figure 12, which depicts a combination of several sub-modules. R-ViSim was implemented in the Python language and using the Python 2D plotting library (Matplotlib). As we can see from Figure 12 that R-ViSim first reads a failure data file (including the detection time and priority of each fault) for rate-based simulation, and R-ViSim allows the user to set the simulation parameters. Once the parameters have been entered, the user can start the simulation by clicking the Simulate button on the menu bar. The proposed model and simulation results are then reported and displayed.

Figure 13 shows a typical window dump from R-ViSim. The high and low fault detection rates, fault correction rate, and the number of debuggers are explained and illustrated

TABLE 11. Comparison results between modeling and simulation for DS3.

Performance Measures	Model-Based Method [9]	Simulation-Based Method	Relative Error
Avg. high priority waiting queue length	1.604×10^{-3}	0	-
Avg. low priority waiting queue length	15.890	15.907	0.11%
Avg. high priority waiting time	1.497×10^{-3}	0	-
Avg. low priority waiting time	7.130	7.123	-0.09%
Avg. high priority response time	1.819	1.818	-0.03%
Avg. low priority response time	8.949	8.943	-0.06%

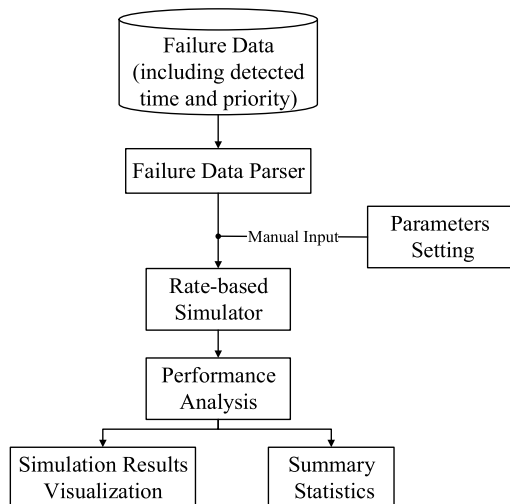


FIGURE 12. High level architecture of the developed R-ViSim tool.

in Section IV-B – IV-D. The length of iteration time can be set to a constant. The right hand side shows the workspace in which the simulation parameters are displayed, while the left hand side shows the plots of these simulation results. The menu items on the top left hand side include the **File** option which allows for file operations, **Reset parameters** option which allows the user to reset all parameters, **Simulate** option which allows the users to start the simulation, **Setting** option provides options for setting all simulation parameters, and **Help** option would provide on-line assistance in case the user needs it.

F. EXPERIMENTAL VALIDITY

The degree of the validity of the experimental design is as important as the experimental results [71], [72]. Adequate validity indicates the experimental results or other findings of the study have a high ability to solve similar problems. In the following, we will briefly discuss the internal, external, and construct validity of the experiment, respectively.

Internal validity refers to whether the research design can correctly explain the research results or show the causal relationship between independent and dependent factors. First, the collection of failure data is one threat to our internal validity. In general, there are not so much failure data containing the information about the fault detection, the fault correction, the fault priority level, etc. In this paper, DS1 and DS2 were

obtained from Bugzilla. DS3 was collected from a projector firmware project developed by Coretronic Corp. in Taiwan. Note that Bugzilla has an open-fault record and its dataset is widely used in software engineering research. Thus we would be able to use Bugzilla’s dataset for doing research. Basically, the three failure data represent a wide variety of applications. Note that the potential inaccuracy of collected failure data is also a threat to internal validity. In actuality, the fault reports from the system could be duplicate, invalid, or unreasonable since OSS projects were usually included thousands of volunteer participants. In this paper, we have carefully inspected the collected data and eliminated the overlaps and invalid bugs from our data sets.

Additionally, the SRGMs we selected for performance comparison are another issue for the internal validity of these experiments. We use some traditional SRGMs introduced in Section II for comparison with our proposed method. These SRGMs have gained wide acceptance in field of software reliability modeling. We also implemented all selected models very carefully.

External validity focuses on the generalizability of the outcome of our work. In other words, whether the experimental results of the research can be applied to other situations. An important threat to the external validity of our experiments is the choice of data sets. In general, we would not be able to generalize our findings and experimental results to every published data set. However, in this paper we totally use three real failure data to evaluate the performance of our proposed method compared with some SRGMs. DS1-DS3 were collected from different OSS and CSS projects, which had different architectures and the different developer compositions. It is also worth noting that Bugzilla has for many years seen widespread use by many users and researchers. It would be more convincing if we perform the experiments based on real failure data collected from the commonly used OSS projects. This diversity allows us to have greater confidence in our experimental results. In this case, we would be able to reduce the threats to external validity.

Construct validity is concerned with whether the measurement can meaningfully reflect or access the underlying construct which is intended to be measured. Thus the construct validity of this study could be affected by the number of comparison criteria in the experiment. In order to check the performance of our proposed method, make a fairly comprehensive comparison with all selected models, and avoid bias, we use three criteria in our paper; they are: the *Theil’s*

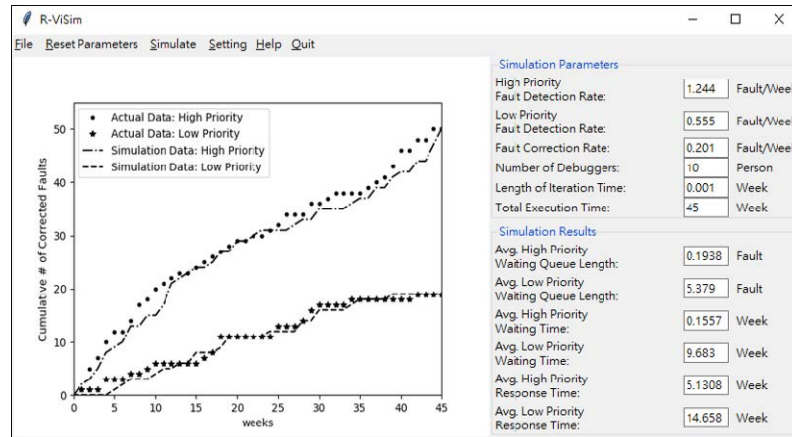


FIGURE 13. A window dump from R-ViSim.

U Statistic (TS), the *Coefficient of Determination* (R^2), and the *Mean Square Error* (MSE). It is worth noting that the TS cannot distinguish between under- or over-prediction, but the magnitude of error can be examined from the computed values of U1 and U2 [69], [70]. MSE is one of the criteria we adopted since it is the primary measure used for comparing prediction methods for a long time. In actuality, MSE is a very common tool for assessing the quality of global model. It is also noticed that the MSE was used to judge the *Retrodictive* ability [3], [4], [8]. On the other hand, the R^2 is a measure of the linear association between the two variables x and y , and the value is bound in the range of 0 to 1. Basically the three criteria may be independent of each other. But it has to be noted that how many of comparison criteria are satisfied and the examination of statistical properties of measures could be another topic of future research [73].

G. RESEARCH QUESTIONS

It is recommended to evaluate a case study by answering some research questions [74]. In this section, we will briefly present the experimental results that answer four research questions.

RQ1: Is there any data-format limitation for the proposed simulation method?

In the field of software reliability engineering, software failure data collections can be classified into two types: interval-domain format and time-domain format [2], [3], [4], [8]. The interval-domain data is characterized by counting the number of failures occurring during a fixed period while the time-domain data records the individual occurrence time of failures. In this paper, DS1-DS3 were collected in interval domain format. In the industry, failure data are usually treated as confidential information. Only few data sets with this feature are released for public access in the literatures. However, most of them belong to interval-domain format. To our knowledge, few time-domain data sets provide the failure correction data in the time-domain format. Considering SRGMs, some may be suitable for one specific format

only, while others apply to the other format. If we want to use the failure data in time-domain format for estimation, our proposed simulation procedures will give users/developers more flexibility in meeting the time-domain format.

Q2: What if a more complicated queuing mechanism is used?

In this paper, based on some assumptions in Section III, we develop simulation procedures that describe the fault removal process, $M/M/c/\infty/PR$. Basically, our proposed simulation procedures contain three steps. The first step is to assign a fault to a free debugger. The second step is used to determine whether a fault is detected according to the failure data set. The end of an iteration is the step of fault correction, which checks whether the faults are completely corrected. Besides the basic queuing characteristics, it is possible for the queuing-based simulation method to involve multistaging or feedback. It can be found that the debugging process depicted in this paper takes into account the fault detection and the fault correction. In fact, a debugging process can typically be subdivided into three steps: fault detection, fault isolation (identifying the location of the root cause of the problem), and fault correction [75]. If the debugger fails to correct an isolated fault for too long, the fault may be sent back for re-isolation (i.e. re-identifying the root cause). Through the use of a multistage queuing system with feedback, the fault isolation and re-isolation in debugging processes can be specified, which will approach reality more closely. It is true that a more complicated queuing system can handle more realistic situations, but further discussion of this subject is beyond the scope of this paper. We plan to study this part and publish our findings in the near future.

RQ3: What is the difficulty of the parameter setting in the simulation method?

In this paper, we have developed a R-ViSim tool to provide quantitative estimation and allow to visualize simulation results. After entering the parameters, user can start the simulation and see the results in a short time. Different parameter values in our proposed simulation method can be executed

and compared with other models and method soon. Thus we will be able to overcome the difficulty of the parameter setting problem of this work and our results can be more easily and closely replicated.

RQ4: What are the benefits of using simulation method for software reliability estimation?

In addition to model-based approaches in Section II.A, the discrete-event simulation generally offers an alternative to analytical models as it can represent the impact of different strategies that may be used during testing [4]. Simulation approaches can relax certain unreasonable assumptions which are common in model-based approaches. It can be seen that our proposed simulator can describe the process of fault correction in detail and optimize various system parameters. Developers and project managers can use the proposed simulation procedure to estimate workloads that are similar to the actual situation and allocate appropriate human resources. Here we will discuss and illustrate how to apply both PPQ simulation procedure and/or PPQ model [9], to project management. Let's assume that the work efficiency of each debugger is equal for convenience of analysis. Due to the limits of paper size, here we will primarily choose DS1 to illustrate how the proposed method can help project managers accurately estimate the efficiency of fault corrections. We can follow similar procedures to make data analysis and discussion for DS2 and DS3.

From Section IV.B, we obtain the average detection rates of high and low-priority faults, i.e., $\lambda_h = 1.244$ and $\lambda_l = 0.555$ faults per week, respectively. To ensure system stability ($\rho = (\lambda/c\mu) < 1$), we assume that the service rate for each debugger μ is 0.201 faults per week, and the number of debuggers c is 10. In this case, we obtain $\rho = (\lambda_h + \lambda_l)/c\mu = 0.895$ as the percentage of faults worked on by each debugger during the testing phase. This analysis shows that we use almost all resources (about 90%), thus introducing risk into the project schedule. Obviously, if any incidents were to happen, such as a debugger's absence or a change in the requirements, a delay in the project schedule may occur. One way to save personnel resources is to delay the release time. For example, with system DS1, if the release schedule is set delayed for a time period roughly equal to one fourth of the time period of the original schedule, which is a delay of 11 weeks, the extended fault detection rate and correction rate are $\lambda' = [(1.244+0.555) \times 45]/(45+11) = 1.45$ faults per week and $\mu' = (2.01*45)/(45+11) = 1.61$ faults per week. The ratio $\rho < 1$, showing that all faults can be corrected by the time the software is released. Compared to not extending the schedule, the personnel resource savings are $(100/2.01) \times (2.01-1.61) = 19.9\%$. That is to say, if 10 debuggers are participated, we can assign at most 2 debuggers to another project or make a better deal with a holiday that might occur within the development schedule.

V. CONCLUSION

Because the reliability of software is critical, software quality evaluation is important. Thus, among those factors typically

used as measurements of software quality, software reliability is regarded as key. A large number of SRGMs have been proposed and discussed over the past three decades. But most of published SRGMs assumed that detected faults will be immediately fixed and/or removed during software testing and debugging. However, most of developers must spend time analyzing the root cause of faults in the real world.

Presently, queueing models have been shown to be useful in many applications. Some studies have shown that queueing theory and/or queueing model can be used to describe various engineering activities for software development, such as testing, debugging, maintenance, etc. In this paper, we propose to use the queueing-based simulation to describe the behavior of FCP and assess the software reliability instead of using model-based approaches. The proposed simulation-based method is therefore inspired by the process scheduling of operating systems and considers the priority levels of the faults. Each fault is assigned a priority level by the testers and faults with higher priority are corrected sooner than faults with lower priority. We thoroughly investigate the fault removal process considering the fault correction time and the finite debugging resources. Three real data sets collected from OSS and CSS are used to test the performance of our proposed method. Experimental results show that the simulation-based method gives a better fit to the observed data than traditional SRGMs and predicts future behavior well. A tool called R-ViSim is developed to automate the simulation task.

On the other hand, Figure 8 shows the difference between detected and removed faults are the open-remaining faults. Practically, the number of open-remaining faults is very important information for managers to allocate adequate debuggers during the software development. When tracking the trend of the number of open-remaining faults, the interval-domain data are generally more ideal than time-domain data. It should be noticed that the format conversion from time-domain to interval-domain can easily be done without any assumptions and precision loss [3]. Therefore, even though the failure data are collected in the time-domain format, it is better to transform them into the interval-domain format before estimating the staffing needs.

Our future works can be divided into three aspects. First, we plan to add more practical functions to our simulation procedure. These will include determining the optimal version-updating time of software systems and taking imperfect debugging into consideration, in order for project leaders to manage the software development process more efficiently. Second, during our research, we observed that debugging teams not only consider the priority of a detected fault, but also its severity. Critical severity faults cause greater damage to systems than low severity faults. Therefore, in future work, severity will be considered in the fault correction process to estimate and evaluate software reliability.

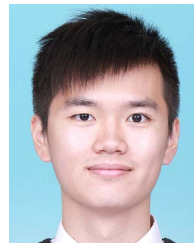
Lastly, we also plan to study another queuing mechanism. In this paper, our proposed queueing-based simulation procedure describes the single-queue scheduling mechanism. In the

future, we plan to use the express queue service in the simulation procedure if the detected faults are not classified according to their level of severity due to some reasons. The detected faults of the software will be delivered to the express queue or allocated to the express queue debuggers if its required service time is below a certain value. Other faults will be delivered to the regular queue or allocated to the regular queue debuggers. In this case, the waiting time can be reduced.

REFERENCES

- [1] *Software Engineering Product Quality—Part 1: Quality Model*, document ISO/IEC 9126-1, 2001.
- [2] H. Pham, *System Software Reliability, Reliability Engineering Series*. London, U.K.: Springer, 2006.
- [3] J. D. Musa, A. Iannino, and K. Okumoto, *Software Reliability, Measurement, Prediction, and Application*. New York, NY, USA: McGraw-Hill, 1987.
- [4] M. R. Lyu, *Handbook of Software Reliability Engineering*. Hightstown, NJ, USA: McGraw-Hill, 1996.
- [5] J. D. Musa, *Software Reliability Engineering: More Reliable Software Faster and Cheaper*, 2nd ed. Bloomington, IN, USA: AuthorHouse, 2004.
- [6] M. Ohba, "Software reliability analysis models," *IBM J. Res. Develop.*, vol. 28, no. 4, pp. 428–443, Jul. 1984.
- [7] R. C. Tausworthe and M. R. Lyu, "A generalized technique for simulation software reliability," *IEEE Softw.*, vol. 13, no. 2, pp. 77–88, Mar. 1996.
- [8] M. Xie, *Software Reliability Modelling*. Singapore: World Scientific, 1991.
- [9] J.-S. Lin, C.-Y. Huang, and C.-C. Fang, "Analysis and assessment of software reliability modeling with preemptive priority queueing policy," *J. Syst. Softw.*, vol. 187, May 2022, Art. no. 111249, doi: 10.1016/j.jss.2022.111249.
- [10] C.-Y. Wu and C.-Y. Huang, "A study of incorporation of deep learning into software reliability modeling and assessment," *IEEE Trans. Rel.*, vol. 70, no. 4, pp. 1621–1640, Dec. 2021.
- [11] K. Z. Yang, "An infinite server queueing model for software readiness assessment and related performance measures," Ph.D. dissertation, Dept. Elect. Eng. Comput. Sci., Syracuse Univ., Syracuse, NY, USA, 1996. [Online]. Available: http://surface.syr.edu/eecs_etd/189
- [12] C.-Y. Huang and C.-T. Lin, "Software reliability analysis by considering fault dependency and debugging time lag," *IEEE Trans. Rel.*, vol. 55, no. 3, pp. 436–450, Sep. 2006.
- [13] C.-Y. Huang and W.-C. Huang, "Software reliability analysis and measurement using finite and infinite server queueing models," *IEEE Trans. Rel.*, vol. 57, no. 1, pp. 192–203, Mar. 2008.
- [14] T. Z. Kuo, "Reliability analysis and application of using finite server queueing models in the detection and removal process of software faults," M.S. thesis, Computer Science Dept., Nat. Tsinghua Univ., Hsinchu, Taiwan, 2013.
- [15] S. Inoue and S. Yamada, "A software reliability growth modeling based on infinite server queueing theory," in *Proc. 9th ISSAT Int. Conf. Rel. Quality Design (QRD)*. Waikiki, HI, USA, Aug. 2003, pp. 305–309.
- [16] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and Mozilla," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 3, pp. 309–346, Jul. 2002.
- [17] H. Zhang, L. Gong, and S. Versteeg, "Predicting bug-fixing time: An empirical study of commercial software projects," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, May 2013, pp. 1042–1051.
- [18] A. Silberschatz, P. Galvin, and G. Gagne, *Operating System Concepts*, 1st ed. Hoboken, NJ, USA: Wiley, 2014.
- [19] H. Deitel, P. Deitel, and D. Choffnes, *Operating Systems*, 1st ed. Upper Saddle River, NJ, USA: Pearson, 2007.
- [20] W. Jones and D. Gregory, "Infinite-failures models for a finite world: A simulation study of fault discovery," *IEEE Trans. Rel.*, vol. 43, no. 2, pp. 211–230, Sep. 2004.
- [21] A. L. Goel and K. Okumoto, "Time-dependent error-detection rate model for software reliability and other performance measures," *IEEE Trans. Rel.*, vol. R-28, no. 3, pp. 206–211, Aug. 1979.
- [22] J. T. Duane, "Learning curve approach to reliability monitoring," *IEEE Trans. Aerosp.*, vol. AS-2, no. 2, pp. 563–566, Apr. 1964.
- [23] A. L. Goel, "Software reliability models: Assumptions, limitations, and applicability," *IEEE Trans. Softw. Eng.*, vol. SE-11, no. 12, pp. 1411–1423, Dec. 1985.
- [24] S. Yamada and S. Osaki, "Reliability growth models for hardware and software systems based on nonhomogeneous Poisson process: A survey," *Microelectron. Rel.*, vol. 23, no. 1, pp. 91–112, Dec. 1983.
- [25] M. Ohba, "Inflection S-shaped software reliability growth model," in *Stochastic Models in Reliability Theory*. Berlin, Germany: Springer-Verlag, 1984, pp. 144–162.
- [26] S. Yamada, M. Ohba, and S. Osaki, "S-shaped reliability growth modeling for software error detection," *IEEE Trans. Rel.*, vol. R-32, no. 5, pp. 475–484, Dec. 1983.
- [27] S. Yamada and S. Osaki, "Software reliability growth modeling: Models and applications," *IEEE Trans. Softw. Eng.*, vol. SE-11, no. 12, pp. 1431–1437, Dec. 1985.
- [28] S. Yamada, S. Osaki, and H. Narihisa, "A software reliability growth model with two types of errors," *RAIRO Oper. Res.*, vol. 19, no. 1, pp. 87–104, 1985.
- [29] S. Yamada and S. Osaki, "Software reliability analysis by considering fault dependency and debugging time lag," *IEEE Trans. Softw. Eng.*, vol. SE-11, no. 12, pp. 1431–1437, Aug. 1985.
- [30] P. K. Kapur and R. B. Garg, "A software reliability growth model for an error-removal phenomenon," *Softw. Eng. J.*, vol. 7, no. 4, p. 291, 1992.
- [31] N. F. Schneidewind, "Fault correction profiles," in *Proc. 14th Int. Symp. Softw. Rel. Eng. (ISSRE)*, Nov. 2003, pp. 257–267.
- [32] P. Kapur, R. Garg, and S. Kumar, *Contributions to Hardware and Software Reliability*. Singapore: World Scientific, 1999.
- [33] Y. F. Hou, "Using the methods of statistical data analysis to improve the trustworthiness of software reliability modeling," M.S. thesis, Comput. Sci. Dept., National Tsinghua Univ., Hsinchu, Taiwan, 2017.
- [34] Z. Liu and R. Kang, "Imperfect debugging software belief reliability growth model based on uncertain differential equation," *IEEE Trans. Rel.*, vol. 71, no. 2, pp. 735–746, Jun. 2022.
- [35] R. Garg, S. Raheja, and R. K. Garg, "Decision support system for optimal selection of software reliability growth models using a hybrid approach," *IEEE Trans. Rel.*, vol. 71, no. 1, pp. 149–161, Mar. 2022.
- [36] M. Nafreen and L. Fiondella, "Software reliability models with bathtub-shaped fault detection," in *Proc. Annu. Rel. Maintainability Symp. (RAMS)*, May 2021, pp. 1–7.
- [37] P. S. Sabnis and S. D. Joshi, "An architecture to enhance, optimize and validate software reliability using machine learning: A contemplate solution," in *Proc. IEEE World Conf. Int. J. Speech Technol. Comput. (AIC)*, Jun. 2022, pp. 21–26.
- [38] H. Yang, L. Yang, N. Hu, Y. Pan, X. Wu, and G. Nie, "Reliability simulation evaluation technology of network system with hardware and software combined," *Proc. Asia Conf. Algorithms, Comput. Mach. Learn. (CACML)*. Hangzhou, China, Mar. 2022, pp. 373–378.
- [39] C. T. Lin and C. Y. Huang, "Quantifying the influences of imperfect debugging on software development using simulation approach," in *Proc. Int. Conf. Adv. Softw. Eng. Appl., Special Session Adv. Technol. Softw. Rel. Saf. (ATSRS)*. Jeju Island, South Korea, Dec. 2009, pp. 305–312.
- [40] E. S. Raymond, *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. Sebastopol, CA, USA: O'Reilly & Associates, 1999.
- [41] D. Gross and C. Harris, *The Fundamentals of Queueing Theory*, 3rd ed. Hoboken, NJ, USA: Wiley, 1988.
- [42] S. S. Gokhale and R. E. Mullen, "Queueing models for field defect resolution process," in *Proc. 17th IEEE Int. Symp. Softw. Rel. Eng. (ISSRE)*. Raleigh, NC, USA, pp. 353–362, Nov. 2006.
- [43] P. K. Kapur, A. G. Aggarwal, and R. Kumar, "A unified approach for discrete software reliability growth model for faults of different severity using infinite server queueing model," *Commun. Dependability Quality Manag. Int. J., Special Guest Issue Trends Future Directions Qual. Rel. Infocom Technol.*, vol. 13, no. 4, pp. 66–81, Dec. 2010.
- [44] P. K. Kapur, S. Anand, S. Inoue, and S. Yamada, "A unified approach for developing software reliability growth model using infinite server queueing model," *Int. J. Rel., Quality Safety Eng.*, vol. 17, no. 5, pp. 401–424, Oct. 2010.
- [45] T. Dohi, S. Osaki, and K. Trivediy, "An infinite server queueing approach for describing software reliability growth—Unified modeling and estimation framework," in *Proc. 11th Asia-Pacific Softw. Eng. Conf. (APSEC)*. Busan, South Korea, Dec. 2004, pp. 110–119.
- [46] N. Zhang, "Queue-based FDP and FCP analysis with detection effort and correction effort," *J. Inf. Comput. Sci.*, vol. 12, no. 1, pp. 21–29, Jan. 2015.
- [47] C.-Y. Huang, T.-Y. Hung, and C.-J. Hsu, "Software reliability prediction and analysis using queueing models with multiple change-points," in *Proc. 3rd IEEE Int. Conf. Secure Softw. Integr. Rel. Improvement*, Jul. 2009, pp. 212–221.

- [48] N. Zhang, G. Cui, and H. Liu, "Software reliability analysis using queueing-based model with testing effort," *J. Softw.*, vol. 8, no. 6, pp. 1301–1307, Jun. 2013.
- [49] M. I. Kellner, "Software process modeling support for management planning and control," in *Proc. 1st Int. Conf. Softw. Process*. Redondo Beach, CA, USA, Oct. 1991, pp. 8–28, doi: 10.1109/ICSP.1991.664337.
- [50] D. M. Raffo and M. I. Kellner, "Empirical analysis in software process simulation modeling," *J. Syst. Softw.*, vol. 53, no. 1, pp. 31–41, Jul. 2000.
- [51] D. Raffo, T. Kaltio, D. Partridge, K. Phalp, and J. F. Ramil, "Empirical studies applied to software process models," *Empirical Softw. Eng.*, vol. 4, pp. 353–369, Dec. 1999.
- [52] S. S. Gokhale and M. R.-T. Lyu, "A simulation approach to structure-based software reliability analysis," *IEEE Trans. Softw. Eng.*, vol. 31, no. 8, pp. 643–656, Aug. 2005.
- [53] I. Rus, J. Collofello, and P. Lakey, "Software process simulation for reliability management," *J. Syst. Softw.*, vol. 46, nos. 2–3, pp. 173–182, Apr. 1999.
- [54] C. T. Lin and C. Y. Huang, "Staffing level and cost analyses for software debugging activities through rate-based simulation approaches," *IEEE Trans. Rel.*, vol. 58, no. 4, pp. 711–724, Dec. 2009.
- [55] A. Juan, J. Faulin, J. Marques, and M. Sorroche, "J-SAEDES: A Java-based simulation software to improve reliability and availability of computer systems and networks," in *Proc. Winter Simulation Conf.*, Dec. 2007, pp. 2285–2292.
- [56] S. S. Gokhale, M. R. Lyu, and K. S. Trivedi, "Incorporating fault debugging activities into software reliability models: A simulation approach," *IEEE Trans. Rel.*, vol. 55, no. 2, pp. 281–292, Jun. 2006.
- [57] G. Antoniol, A. Cimitile, G. A. D. Lucca, and M. D. Penta, "Assessing staffing needs for a software maintenance project through queueing simulation," *IEEE Trans. Softw. Eng.*, vol. 30, no. 1, pp. 43–58, Jan. 2004.
- [58] W. Fan, Y. Xiaohu, Z. Xiaochun, and C. Lu, "Simulation of the defect removal process with queueing theory," in *Proc. 3rd Int. Symp. Empirical Softw. Eng. Meas.*, Oct. 2009, pp. 473–476.
- [59] S. C. Chang, C. Y. Huang, and J. S. Lin, "Applying express-queue-based approach to software reliability and cost analysis," in *Proc. IEEE Int. Conf. Signal Process., Commun. Comput. (ICSPCC)*. Hong Kong, Aug. 2016, pp. 1–6.
- [60] C.-T. Lin and Y.-F. Li, "Rate-based queueing simulation model of open source software debugging activities," *IEEE Trans. Softw. Eng.*, vol. 40, no. 11, pp. 1075–1099, Nov. 2014.
- [61] Y. Shu, Z. Wu, H. Liu, and Y. Gao, "A simulation-based reliability analysis approach of the fault-tolerant web services," in *Proc. 7th Int. Conf. Intell. Syst., Modeling Simulation (ISMS)*, Jan. 2016, pp. 125–129.
- [62] H. Nakahara, A. Monden, and Z. Yucel, "A simulation model of software quality assurance in the software lifecycle," in *Proc. IEEE/ACIS 22nd Int. Conf. Softw. Eng., Artif. Intell., Netw. Parallel/Distrib. Comput. (SNPD)*, Nov. 2021, pp. 236–241.
- [63] L. Kleinrock, *Queueing Systems*, 1st ed. Hoboken, NJ, USA: Wiley, 2016.
- [64] A. B. Bondi and J. P. Buzen, "The response times of priority classes under preemptive resume in M/G/m queues," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 12, no. 3, pp. 195–201, Aug. 1984.
- [65] I. Samoladas, L. Angelis, and I. Stamelos, "Survival analysis on the duration of open source projects," *Inf. Softw. Technol.*, vol. 52, no. 9, pp. 902–922, Sep. 2010.
- [66] K. S. Trivedi, *Probability and Statistics With Reliability, Queuing, and Computer Science Applications*, 2nd ed. Hoboken, NJ, USA: Wiley, 2002.
- [67] *Bugzilla*. Accessed: Apr. 30, 2016. [Online]. Available: <https://bugzilla.mozilla.org/>
- [68] *Eclipse Bugzilla*. Accessed: Mar. 12, 2016. [Online]. Available: <https://bugs.eclipse.org/bugs/>
- [69] G. Keller and B. Warrack, *Statistics for Management and Economics*. Duxbury, MA, USA: Duxbury Press, 1999.
- [70] K. Holden, D. A. Peel, and J. L. Thompson, *Economic Forecasting: An Introduction*. Cambridge, U. K.: Cambridge Univ. Press, 1991.
- [71] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Berlin, Germany: Springer, 2012.
- [72] M. V. Zelkowitz and D. Wallace, "Experimental validation in software engineering," *Inf. Softw. Technol.*, vol. 39, no. 11, pp. 735–743, 1997.
- [73] S. D. Conte, H. E. Dunsmore, and V. Y. Shen, *Software Engineering Metrics and Models*. Redwood City, CA, USA: Benjamin-Cummings, 1986.
- [74] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Softw. Eng.*, vol. 14, no. 2, pp. 131–164, Apr. 2009.
- [75] R. S. Pressman and B. Maxim, *Software Engineering: A Practitioner's Approach*, 9th ed. New York, NY, USA: McGraw-Hill, 2019.



JHih-Sin Lin received the B.S. degree in computer science from the National Central University, Taoyuan, Taiwan, in 2015, and the M.S. degree in computer science from the National Tsing Hua University, Hsinchu, Taiwan, in 2017. He is currently an Engineer at Silicon Motion, Inc., Zhubei, Hsinchu. His current research interests include software reliability estimation and quality measurement.



Chin-Yu Huang (Member, IEEE) received the M.S. and Ph.D. degrees in electrical engineering from the National Taiwan University, Taipei, in 1994 and 2000, respectively. He is currently a Full Professor with the Department of Computer Science and the Institute of Information Systems and Applications, National Tsing Hua University (NTHU), Hsinchu, Taiwan. He was with the Bank of Taiwan, from 1994 to 1999. He was also a Senior Software Engineer at Taiwan Semiconductor Manufacturing Company, from 1999 to 2000. Before joining NTHU, he was a Division Chief at Central Bank of China, Taipei, in 2003. His research interests include software reliability engineering, software testing, software metrics, software testability, fault tree analysis, and system safety assessment. He received the Ta-You Wu Memorial Award of National Science Council, Taiwan, in 2008. He has been on the Editorial Board of *Scientific Programming*, since 2017 and the *Journal of Information Science and Engineering*, since 2016. He is currently serving as an Associate Editor for the IEEE TRANSACTIONS ON RELIABILITY.

• • •