Received 12 April 2023, accepted 18 May 2023, date of publication 23 May 2023, date of current version 1 June 2023.
Digital Object Identifier 10.1109/ACCESS.2023.3279109

RESEARCH ARTICLE

Using Modified Diffusion Models for Reliability Estimation of Open Source Software

KUAN-JU CHEN[1] AND CHIN-YU HUANG[2], (Member, IEEE)
[1]Garmin, Taoyuan 333, Taiwan
[2]Department 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 National Science and Technology Council, Taiwan, under Grant MOST 110-2221-E-007-035-MY3 and Grant MOST 111-2221-E-007-079-MY3.

ABSTRACT Software development is a highly unpredictable process, and ensuring software quality and reliability before releasing it to the market is crucial. One of the common practices during software development is the reuse of code. It can be achieved by utilizing libraries, frameworks, and other reusable components. Practically, when a fault is detected in replicated code, developers must check for similar faults in other copies, as there is a dependency between faults. To prevent recurrence of observed failures, developers must remove the corresponding leading fault and any related dependent faults. Many software reliability growth models (SRGMs) have been proposed and studied in the past, but most SRGMs assume that developers usually detect only one fault causing a failure. In actuality, it is necessary to consider the possibility of detecting multiple faults that may share similarities or dependencies. Additionally, some SRGMs rely on specific assumptions that may not always be valid, such as perfect debugging and/or immediate debugging. In this study, the modified diffusion models are proposed to handle these unrealistic situations, and are expected to better capture the dynamics of open source software (OSS) development. Experiments using real OSS data show that the proposed models can accurately describe the fault correction process of OSS. Finally, an optimal software release policy is proposed and studied. This policy takes into account some factors, including the remaining number of faults in the software, the expenses associated with identifying and rectifying those faults, and the level of market demand for the software. By considering these factors, developers can determine the optimal time to release the software to the market.

INDEX TERMS Software reliability, diffusion model, open source software, software release, debugging, testing.

I. INTRODUCTION

In today's technological landscape, software plays a critical role in numerous safety- and/or life-critical systems that undergo an extremely rigorous certification process. The primary concern, however, is how to produce high-quality and reliable software products as quickly as possible. Typically, the source code of software systems under development can be divided into two categories: closed-source software (CSS) systems like Microsoft Office, Apple's iOS, and others, and open-source software (OSS) systems like Firefox, Ubuntu Linux, Apache, GCC, and others. To assess the reliability of both CSS and OSS systems, software practitioners must monitor the failure occurrence process using collected failure data. With the accumulation of more failure data or customer-reported bugs, it becomes possible to more accurately predict software reliability, failure intensity, the total number of initial faults, the remaining faults, and other parameters.

In software testing, assessing software reliability is crucial as it helps developers make informed decisions throughout the software development process. Reliability is often mathematically defined as the probability that a system or a system capability functions without failure for a specified duration or number of natural units within a given environment [1].

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

Essentially, reliability is dependent on various characteristics of the software product and development process. Over the past three decades, there have been numerous *Software Reliability Growth Models* (SRGMs) published, each with its own set of assumptions, strengths, and weaknesses [1], [2], [3]. Many SRGMs consider different development environments and operational conditions when predicting software reliability. Software reliability engineering is firmly grounded in a well-established theoretical framework that encompasses operational profiles, SRGMs, statistical estimation, and sequential sampling theory [1], [2]. Kim et al. [4] once proposed a software reliability engineering process for software development in the Korea defense industry. The process involves incorporating additional reliability activities into each phase of the software development lifecycle, along with the application of models and metrics for assessing and analyzing software reliability. By analyzing the failure data collected during testing, SRGMs can be utilized to make predictions about software reliability.

It is worth noting that SRGMs not only serve as valuable tools for testing, but they can also be utilized for debugging purposes. Practically, software development often involves many different parts of the software that perform similar tasks. To increase efficiency, programmers typically reuse existing code rather than rewriting similar code from scratch. However, this practice of copying and pasting code can lead to the creation of replicated code within the software. When a fault is found in this replicated code, it is common for developers to also search for similar faults in other copies of the code. This phenomenon of fault dependency can pose significant problems for software development. A fault can fall into one of two categories: *leading* or *dependent* faults. Leading faults are those that can be removed directly. Conversely, mutually dependent faults cannot be eliminated until the corresponding leading fault is addressed first.

When a failure occurs, it may be the result of a mutually dependent fault, meaning that developers must address the corresponding leading fault as well as any related dependent faults to prevent future failures from occurring. As a result, developers may need to identify and address multiple faults when investigating a failure. To account for the possibility of detecting multiple faults that may be similar or dependent on each other, we propose the term "additional debugging effort" to reflect the fact in this study. However, it has to be noted that many existing SRGMs do not consider the phenomenon of additional debugging and assume that developers can only identify and eliminate the single fault responsible for a given failure. Therefore, it is necessary to relax some assumptions by accounting for the possibility of multiple faults and additional debugging effort [5].

On the other hand, it can be found that most SRGMs rely on a set of assumptions that may not always be reasonable, particularly in the context of developing OSS. One of these assumptions is immediate and perfect debugging, this is, faults are detected and corrected instantly and flawlessly. However, this assumption is often not true because it ignores the time delay that can occur between fault detection and correction. The design and development process of CSS and OSS differ significantly from a usage pattern perspective. CSS is not only developed within a software company but is also sold on the public market and used by numerous users. In OSS development, debugging can take longer due to the lack of dedicated resources and personnel. There is no guarantee that a detected fault will be resolved immediately, as developers may prioritize different tasks or face resource constraints.

In general, OSS can be characterized by its informality and lack of formal documentation at times [6]. This can make it more challenging for developers to analyze faults, leading to a greater likelihood of imperfect debugging. In this case, traditional SRGMs could not be well-suited for modeling the reliability of OSS since they don't account for the unique challenges and complexities of OSS development [7]. In this study, we propose to extend the diffusion model proposed by Bass [8] to handle the phenomenon of additional debugging. The diffusion model originally was used to predict sales of new consumer products. The primary concept behind the diffusion model in economics is to distinguish buyers into two groups: *innovators* and *imitators*. *Innovators* are individuals who adopt a new product based on their own perceptions and without any external influence. In contrast, *imitators* are individuals whose adoption decisions are influenced by the social pressures of the surrounding system. This model acknowledges the impact of social influence on the adoption of a new product and can be used to predict and analyze the diffusion process of the product in the market.

Likewise, we can categorize removed faults as those that caused failures and those that did not. When a fault is removed, there is a possibility that other related faults can also be removed without causing any further failures. This is because of the presence of fault dependencies and similarities, where removing one fault can reveal and remove other faults as well. This highlights the importance of thorough debugging and testing to identify and remove all faults, including those that may not be immediately apparent but could cause future failures if left unaddressed [9].

Based on the discussions above, we aims to extend the diffusion model to account for the realistic development process of OSS. This includes considering phenomena such as additional debugging, debugging time lags, and imperfect debugging that are commonly observed in practice. Actual data collected from two OSS projects will be used to present and analyze the experiments. Moreover, the proposed modified diffusion model serves as the basis for creating constructive rules to identify the ideal time for software release. These rules can assist project managers in deciding the appropriate moment to conclude testing and launch the software to the market.

The paper is organized as follows. In Section II, we provide a literature survey that covers past works on modifying the assumptions of traditional SRGMs to make them more suitable for software reliability analysis. Section III introduces how the diffusion model can be applied to software reliability and how it can be extended to derive new models that consider the aforementioned phenomena. Section IV presents an evaluation of our models using two OSS projects and compares the results with those obtained from traditional SRGMs. In Section V, we discuss how the proposed models can be used to determine the optimal release time and offer a decision procedure. Finally, Section VI provides our conclusion, where we summarize the contributions of this study.

II. BACKGROUND AND LITERATURE REVIEW

As time progresses and technology continues to develop, software has become an increasingly integral part of our daily lives. It is crucial that software functions correctly for end-users, making the assessment of software reliability a critical step in the testing phase. To accurately predict software reliability, various SRGMs have been proposed to describe the process of fault detection and correction [1], [2], [3]. It is important to note that there are still many SRGMs published in the literature. These models vary in terms of their underlying assumptions, mathematical formulations, and applicability to different types of software systems.

For example, Liu et al. [11] considered software reliability modeling under the framework of uncertainty theory, and deduced a software belief reliability growth model (SBRGM) using uncertain differential equations. Basically, Liu et al.'s SBRGM is based on the concept of belief degrees, which are used to measure the degree of belief that a particular software component will not fail at a given time. The model uses uncertain differential equations to describe the growth of belief degrees over time, taking into account the effects of various factors that can affect software reliability. Their experimental results suggested that the proposed SBRGM outperforms several popular probability-based SRGMs, such as the exponential model, the power law model, and the Weibull model. In addition, Yang et al. [12] once proposed a method for predicting the trend and quantity of bugs in new versions of a software project by combining complex network theory with the panel data model. Their approach is applicable in both within-project and cross-project contexts. Their proposed method leverages the insights from complex network theory to model the relationships between software components and their interactions in a software project. They used this network-based approach to analyze the factors that affect software bug occurrences, including code complexity, software architecture, and development team dynamics.

Vizarreta et al. [13] also developed a framework that utilizes the SRGM to evaluate and anticipate the level of maturity of software-defined networking (SDN) controllers. The goal of their framework is to provide valuable insights into the dependability and functionality of SDN controllers, and to facilitate the identification of potential areas of improvement. They presented some guidelines to assist network operators in making informed decisions about the deployment of SDN controller software in operational environments. Garg et al. [14] proposed a hybrid approach called Entropy-Combinative Distance-Based Assessment (CODAS-E) to effectively select and rank software reliability growth models using multiple performance indexes. In addition, Wu et al. [15] proposed an approach to incorporate the time dependencies between the fault detection, and fault correction processes, focusing on the parameter estimations of the combined model.

Generally, many of the traditional SRGMs are based on similar assumptions [1], [8]. In some cases, faults detected during testing may not be corrected immediately. This delay in fault correction can be caused by various factors such as prioritization of tasks, availability of resources, or complexity of the fault. As a result, the software reliability growth process can be impacted, and traditional SRGMs may not accurately reflect the actual reliability of the software. To address this, some SRGMs have been developed to account for delayed fault correction and other factors that may affect the software reliability growth process [11], [12], [15]. Raymond and O'Reilly [6] noticed that beta testing plays a crucial role in the testing of OSS. Therefore, the test team for OSS is often separate from the development team. When a fault is detected in OSS, developers typically require additional time to communicate with testers and rectify the issue. This situation often leads to longer debugging times for OSS, and delays in the fault correction process are more likely to occur.

The assumption of perfect debugging in the fault correction process is unrealistic as it is impossible for developers to achieve perfect debugging throughout the entire correction process. When trying to fix more complex faults, developers may inadvertently introduce new issues. Furthermore, studies indicate that the development process of open-source software (OSS) lacks structure and formal documentation, including the testing phase. As a result, debugging can be especially challenging for OSS, and traditional SRGMs that assume perfect debugging may not provide an accurate assessment of software reliability [7]. The absence of formalized procedures and documentation, combined with the separation of development and testing teams, creates a challenging and complex debugging process for OSS. As a result, it is common to encounter imperfect debugging, where developers are unable to completely eliminate all faults during the correction process. This phenomenon is a direct consequence of the unique nature of OSS development, which relies heavily on collaboration between teams and individuals. It is noted that Pham [10] has previously shown that models that account for imperfect debugging provide a more accurate fit to actual fault data compared to models that do not consider this factor. In other words, the inclusion of imperfect debugging in the models results in a better representation of the real-world performance of software systems.

To address the issue of time lags between fault detection and correction, various research papers have proposed novel software reliability growth models (SRGMs). For instance, Schneidewind [16] developed a model that treated the fault correction process as a delayed fault detection process, with a fixed delay time assumed for all corrections. However, this assumption of a constant lag time is unrealistic, as the level of difficulty for each fault can vary significantly, and the time taken to correct each fault can also differ. Xie and Zhao [17] addressed the issue of constant delay time in the Schneidewind model by introducing a time-dependent function for the delay time. Xie et al. [18] once presented a new SRGM that accounts for heterogeneous faults. The model assumes that each fault has specific parameters for detection time and correction delay, which can follow arbitrary distributions.

Huang and Lin [19] showed experimentally that there is evidence of fault dependency. In cases where a mutually dependent fault results in a failure, it is imperative for developers to first address the associated leading fault before attempting to remedy the dependent fault. As the form of the fault content function is dependent on testing environments, several different forms have been proposed. Additionally, Yamada et al. [20] used a linearly increasing function and an exponentially increasing function as fault content functions.

It is important to highlight that most SRGMs, including the one mentioned above, do not take into account the possibility of additional debugging due to the interdependence and similarity of faults. These models typically assume that when a failure occurs, only the fault responsible for the failure will be identified. Li et al. [21] found that large software suites often contain a considerable amount of duplicated code, and detecting a fault in one instance of the replicated code can lead to the identification of similar faults in the other instances. For example, the X window system's entire source code comprises 19% of duplicated code [22]. Although copy-and-paste activities can reduce programming efforts, replicated code can introduce faults [21]. This is primarily because programmers may copy and paste code without making necessary modifications.

To account for additional debugging effort, Kapur et al. [9] utilized the diffusion model as their SRGM. The diffusion model, originally proposed by Bass [2] in economics, initially considers the number of potential purchasers as a constant. However, Mahajan and Peterson [23] argued that the number of potential purchasers should vary over time. Similarly, due to imperfect debugging, the fault content should be treated as a time-dependent function rather than a constant. As the phenomenon of additional debugging is prevalent in OSS, we have extended the diffusion model to account for it. Our approach considers the realistic conditions of OSS development, which includes the presence of time-lags in debugging and imperfect debugging. In the following, we propose to develop new diffusion models that more accurately fit the observed fault data in OSS projects.

III. MODELING FAULT CORRECTION PROCESS

A. GENERALIZED DELAYED NHPP RELIABILITY MODEL

Previous research has proposed several models, with many conventional NHPP reliability models being based on the following assumptions [1], [8].

1. The fault detection/correction process is modeled by NHPP.
2. Software is subject to failures during execution caused by faults remaining in the software.
3. The mean number of faults detected in the time interval $(t, t + \Delta t]$ is proportional to the mean number of remaining faults in the system.
4. The failure rate of each detectable fault is identical.
5. Each time a failure is detected, the fault is immediately removed.
6. During the fault detection/correction process, no new faults are introduced into the software.

Based on these assumptions, Pham et al. [8] introduced a generalized NHPP reliability model that can be obtained by solving the following equation:

$$\frac{dm(t)}{dt} = b(t)[a(t) - m(t)], \tag{1}$$

where $m(t)$ is the mean value function (MVF) of the expected number of faults removed in time $(0, t]$, $b(t)$ is a time-dependent fault detection rate function, and $a(t)$ is a time-dependent fault content function. Note that the functions $a(t)$ and $b(t)$ are defined based on certain assumptions regarding the behavior of the detection process. The majority of traditional SRGMs assume perfect debugging, which leads to the assumption that $a(t)$ is a constant. After solving (1) under the initial condition $m(0) = 0$, we can get the generalized NHPP SRGM as:

$$m(t) = e^{-B(t)} \int_0^t a(\tau)b(\tau)e^{B(\tau)}d\tau, \tag{2}$$

and

$$B(t) = \int_0^t b(\tau)d\tau. \tag{3}$$

To account for time lags between fault detection and correction, a generalized delayed NHPP reliability model is derived by extending (2). This is necessary because (2) does not take into consideration the possibility of such time lags.

$$m_{new}(t) = m(t - \varphi(t)) = e^{-B(t-\varphi(t))} \int_0^{t-\varphi(t)} a(\tau)\, b(\tau)\, e^{B(\tau)}d\tau, \tag{4}$$

The delayed-time function $\varphi(t)$ is used to describe the time lags between fault detection and correction. In many studies, it is assumed that when a failure is detected, the fault is immediately removed, resulting in $\varphi(t)$ being equal to 0.

As previously mentioned, Assumption (5) and (6) are not realistic in practice, and they assume that $a(t) = a$ and $\varphi(t) = 0$. In the following section, we propose the use of the diffusion model for software reliability analysis. We begin by deriving the basic diffusion model, which only accounts for additional debugging. We then relax some of the assumptions that are not reasonable in practice and derive the extended diffusion models, which consider the additional debugging effort, debugging time lags, and imperfect debugging.

B. APPLYING A DIFFUSION MODEL IN SOFTWARE RELIABILITY MODELING

The diffusion model shares many of the underlying assumptions of conventional SRGMs. However, it differs in one significant aspect: *Upon a failure observation, the fault detection phenomenon may also detect the proportion of the remaining faults.*As previously mentioned, the occurrence of a single failure may lead to the discovery of not only the fault responsible for the failure but also related or similar faults. This phenomenon is known as the diffusion process, which is described by the exclusive assumption mentioned above. Using the assumptions of perfect debugging and immediate debugging, we can define the fault content function as follows:

$$a\,(t) = a, \tag{5}$$

and the delayed-time function as:

$$\varphi\,(t) = 0. \tag{6}$$

Moreover, due to the above exclusive assumption for the diffusion model, the time-dependent fault detection rate is:

$$b\,(t) = \left(p + q\frac{m\,(t)}{a}\right), \tag{7}$$

where $m(t)$ is the mean value function (MVF) of the expected number of the faults detected in time $(0, t]$, a is the number of initial faults in the software, p is the failure rate, and q is the additional fault detection rate. Since the expected number of faults detected at time 0 always is zero, we can get the following MVF by substituting (7) and (5) into (1) and solving it under the initial condition $m\,(t) = 0$:

$$m\,(t) = a\frac{1 - e^{-(p+q)t}}{1 + \frac{q}{p}e^{-(p+q)t}}. \tag{8}$$

Basically the diffusion model that is derived is highly flexible, allowing for the modeling of software reliability growth in various ways. This includes the ability to shape the growth curve as either S-shaped or exponential by utilizing different combinations of p and q. If $p \geq q$, the shape will be exponential, otherwise it will be S-shaped. It is worth noting that if only one fault is detected for each failure, i.e., $q = 0$, then this model will reduce to a Goel-Okumoto (GO) model [6]. Because (7) includes $m(t)$, we cannot derive the MVF by directly substituting it into (4). In order to get the extended diffusion models from the generalized delayed NHPP reliability model in (4), we have to use a new modeling approach for the diffusion process to avoid complicating the resultant models [20].

The diffusion model is a flexible approach to software reliability modeling, as it allows the fault detection rate to be described using various distribution functions. These functions can help account for the complexity of software development and the variability in the fault detection process [23]. To avoid mathematical complexity, we will propose a new form that takes into account the diffusion process to describe the fault detection rate. Here, we use the logistic as the fault detection rate proposed by Pham et al. [24]:

$$b(t) = \frac{p + q}{1 + \frac{q}{p}e^{-(p+q)t}}. \tag{9}$$

Substituting (9) into (4) with constant values of $a(t)$ and $\varphi(t)$ as a and 0, respectively, we have

$$m(t) = a\frac{1 - e^{-(p+q)t}}{1 + \frac{q}{p}e^{-(p+q)t}}. \tag{10}$$

It is evident that (10) is equivalent to (8). Thus, we can demonstrate that the logistic form of $b(t)$ can describe the diffusion process in the same way as the original form, (7). Therefore, to simplify the mathematics involved and develop extended diffusion models, we will use (9) as the fault detection equation for the diffusion model in the subsequent section.

C. MODELING OF DEBUGGING TIME LAGS AND IMPERFECT DEBUGGING IN DIFFUSION MODELS

Complicated faults can lead to a time lag between detection and correction, particularly in OSS development. Practivally, beta testing is a vital part of OSS testing [3], necessitating time for developers to analyze failure reports and communicate with testers. It is evident that a time lag exists between fault detection and correction in OSS. To account for this reality, we modify Assumption (5) presented in Section III-A as follows:

The detected fault may not be immediately removed, and follows the fault detection process with a time lag$\varphi(t)$.

It is important to note that modifying Assumption (5) in Section III-A is inadequate since it assumes not only immediate debugging, but also that a single failure is caused by only one fault, and the primary fault will be immediately removed, resulting in the failure occurring only once [23]. After modifying Assumption (5) in Section III-A, we also need to introduce an additional assumption as follows:

Only the first occurrence of a failure is counted and each failure is caused by exactly one fault.

The accuracy of SRGMs that incorporate the delayed-time function $\varphi(t)$ in describing the correction process in practice is dependent on the specific form of $\varphi(t)$. Previous research [18] suggests that certain existing SRGMs can be interpreted as delayed detection SRGMs based on the GO model, and the delayed-time functions listed below can be extracted from conventional SRGMs.

- Delayed-time Function 1 (DF1) from the Yamada delayed S-shaped model [7], [24]:

$$\varphi_1\,(t) = \frac{1}{p}\ln\left(1 + pt\right). \tag{11}$$

- Delayed-time Function 2 (DF2) from the Inflected S-shaped model [7], [24]:

$$\varphi_2\,(t) = \frac{\psi e^{-pt}}{p\left(1 + \psi e^{-pt}\right)}. \tag{12}$$

Once we extract the delayed-time functions from conventional SRGMs, we will be able to develop diffusion models that not only describe the process of fault correction but also take into account the time lags associated with debugging. These models are useful in predicting the effectiveness of fault correction and can aid in the optimization of debugging processes.

Case 1: Extended diffusion model Model #1 when using DF1. Here we assume:

$$a(t) = a, \tag{13}$$

$$\varphi(t) = \frac{1}{p}\ln(1+pt), \tag{14}$$

$$b(t) = \frac{p+q}{1+\frac{q}{p}e^{-(p+q)t}}. \tag{15}$$

From the generalized delayed NHPP reliability model in (4), we have

$$m(t) = a\frac{1-e^{-(p+q)(t-\frac{1}{p}\ln(1+pt))}}{1+\frac{q}{p}e^{-(p+q)(t-\frac{1}{p}\ln(1+pt))}}. \tag{16}$$

Case 2: Extended diffusion model Model #2 when using DF2. Here we assume:

$$a(t) = a, \tag{17}$$

$$\varphi(t) = \frac{\psi e^{-pt}}{p(1+\psi e^{-pt})}, \tag{18}$$

$$b(t) = \frac{p+q}{1+\frac{q}{p}e^{-(p+q)t}}. \tag{19}$$

From the generalized delayed NHPP reliability model in (4), we have:

$$m(t) = a\frac{1-e^{-(p+q)(t-\frac{\psi e^{-pt}}{p(1+\psi e^{-pt})})}}{1+\frac{q}{p}e^{-(p+q)(t-\frac{\psi e^{-pt}}{p(1+\psi e^{-pt})})}}. \tag{20}$$

As previously mentioned, the debugging process is a complex task that involves identifying and rectifying relevant faults, particularly in the context of OSS development. The process poses a significant challenge for developers due to the unstructured nature of OSS development and the lack of formal documentation, increasing the likelihood of incomplete or imperfect debugging. In order to better accommodate real-world scenarios, we have made modifications to assumption (6) as presented in Section III-A, as follows [24]:

During the debugging phase, a detected/ removed fault will introduce faults at a constant rate r.

In this case, we formulate the time-dependent fault content function $a(t)$ as:

$$a(t) = a(1+rt), \tag{21}$$

where r is the increasing rate of the number of introduced faults to the initial fault. $a(t)$ is the number of total faults, including the initial faults and introduced faults, at time t and $a(t) = a$. Once we have obtained the fault content function in (21), we suggest that the models should also take into account imperfect debugging and debugging time lag to more accurately depict the software correction process in practice.

Case 3: Extended diffusion model Model #3, when using DF1 and imperfect debugging:

$$a(t) = a(1+rt), \tag{22}$$

$$\varphi(t) = \frac{1}{p}\ln(1+pt), \tag{23}$$

$$b(t) = \frac{p+q}{1+\frac{q}{p}e^{-(p+q)t}}. \tag{24}$$

From the generalized delayed NHPP reliability model in (4), we get (25).

$$\begin{aligned} m(t) = {} & \frac{a}{1+\frac{q}{p}e^{-(p+q)\left(t-\frac{1}{p}\ln(1+pt)\right)}} \\ & \times \left(\left(1-e^{-(p+q)\left(t-\frac{1}{p}\ln(1+pt)\right)}\right)\left(1-\frac{r}{p+q}\right)\right. \\ & \left. + r\left(t-\frac{1}{p}\ln(1+pt)\right)\right). \end{aligned} \tag{25}$$

Case 4: Extended diffusion Model #4, when using DF1 and imperfect debugging:

$$\varphi(t) = \frac{\psi e^{-pt}}{p(1+\psi e^{-pt})}, \tag{26}$$

$$b(t) = \frac{p+q}{1+\frac{q}{p}e^{-(p+q)t}}. \tag{27}$$

From the generalized delayed NHPP reliability model in (4), we get (28).

$$\begin{aligned} m(t) = {} & \frac{a}{1+\frac{q}{p}e^{-(p+q)\left(t-\frac{\psi e^{-pt}}{p\left(1+\psi e^{-pt}\right)}\right)}} \\ & \times \left(\left(1-e^{-(p+q)\left(t-\frac{\psi e^{-pt}}{p\left(1+\psi e^{-pt}\right)}\right)}\right)\left(1-\frac{r}{p+q}\right)\right. \\ & \left. + r\left(t-\frac{\psi e^{-pt}}{p\left(1+\psi e^{-pt}\right)}\right)\right). \end{aligned} \tag{28}$$

IV. DATA ANALYSIS AND EXPERIMENTS FOR OSS

A. DATA SETS

In this study, we have selected two widely recognized OSS data sets to assess the accuracy of our proposed models. The first data set (DS1) was obtained from Eclipse 3.7, and was collected from the Bugzilla site of Eclipse [27], [28]. The second data set (DS2) was obtained from Tomcat 5, and was collected from the Bugzilla site of the Apache Software Foundation (ASP) [29]. Table 1 provides a summary of DS1 and DS2. It is worth noting that the cumulative faults curve for DS1 is S-shaped, while that of DS2 follows an exponential pattern. This implies that the data sets we have collected are suitable for both exponential and s-shaped models.

Additionally, we performed a Laplace trend test on the chosen datasets to determine whether the software's reliability

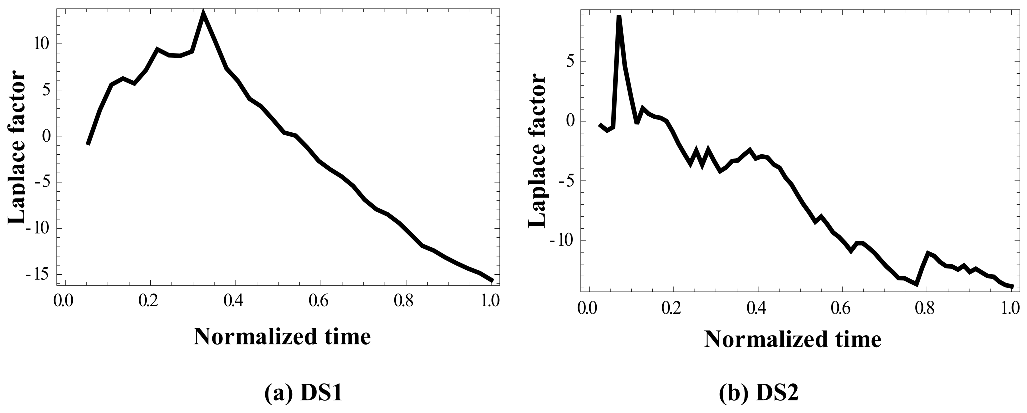


FIGURE 1. Laplace trend test.

TABLE 1. Summary of selected data sets.

Software	Start time	End time	No. of faults	Remarks
Eclipse 3.7	2010/6	2013/10	1,006	It is a cross platform integrated development environment (IDE) and can be used to develop applications [27][28]
Tomcat 5	2004/11	2010/9	754	It is a web server and servlet container, and it provides a pure Java HTTP web server environment for Java code to run in [29]

was increasing. The Laplace trend factor at time period k, $u(k)$, is defined as follows:

$$u(k)=\frac{\sum_{i=1}^{k}(i-1)n(i)-\frac{k-1}{2}\sum_{i=1}^{k}n(i)}{\sqrt{\frac{k^2-1}{12}\sum_{i=1}^{k}n(i)}}, \quad (29)$$

where $n(i)$ is the number of faults removed at time period i. In the fault counts model, we split the total time interval for testing into $n$time periods. The Laplace trend test determined the trend (increase/decrease) of each time period by plugging in time into (30). A positive value of $u(k)$ indicated that the software reliability was decreasing during time period k, while a negative value of $u(k)$ indicated that the software reliability was increasing during time period k. To facilitate comparison, we normalized the testing time intervals of the selected data sets to [0,1].

Fig. 1(a) shows that during the initial 35% of the testing time, the values of $u(k)$ demonstrate an upward trend, which means that the reliability falls. Throughout the remaining 65% of the testing time, there is a persistent decline, suggesting an improvement in reliability. Similarly, upon examination of Fig. 1(b), in the initial 10% of the testing period, $u(k)$ values display an upward trend, indicating a decline in reliability. However, during the remaining 90% of the testing time, there is a consistent decrease in $u(k)$ values, suggesting an improvement in reliability. Fig. 1(a) and (b) illustrates that there are signs of improved reliability after the completion of testing. Therefore, our proposed model and other SRGMs can be utilized to forecast the quantity of detected faults in DS1 and DS2.

B. COMPARISON CRITERIA

To evaluate the performance of all selected models, we employ the following criteria for assessment and to provide quantitative comparisons.

(1) The *Mean Square Error*(MSE) is defined as [1] and [8]:

$$MSE=\frac{1}{n-k}\sum_{i=1}^{n}\left(m\left(t_i\right)-m_{t_i}\right)^2, \quad (30)$$

where $m\left(t_i\right)$ denotes the estimated number of faults removed by time t_i, m_{t_i} denotes the actual number of faults removed by time t_i and n denotes the size of the adopted data set, $k$is the degree of freedom, which means the number of estimated parameters in the adopted SRGM. The MSE is a commonly used criterion to assess the performance of SRGMs. It quantifies the difference between the estimated value and the actual data. A smaller value of the MSE indicates a better fit of the SRGM. It is important to consider that the MSE takes the degree of freedom into account, which means that the number of estimated parameters in the SRGM can also impact the value of the MSE.

(2) The *Coefficient of determination* (R^2) is defined as [30]:

$$R^2=1-\frac{\sum_{i=1}^{n}\left(m\left(t_i\right)-m_{t_i}\right)^2}{\sum_{i=1}^{n}\left(m_{t_1}-\overline{m}\right)^2}. \quad (31)$$

The coefficient of determination, denoted as R^2, is a measure of the goodness of fit of a SRGM to data points. R^2 ranges from 0 to 1, representing the proportion of data variation that the SRGM can account for. As a result, a higher R^2 value indicates a better fit of the SRGM to the data.

(3) The *Theil's U Statistics* are defined by [31] and [32]

$$U1 = \frac{\sqrt{\sum_{i=1}^{n}\left(m_{t_i} - m\left(t_i\right)\right)^2}}{\sqrt{\sum_{i=1}^{n} m_{t_i}^2} + \sqrt{\sum_{i=1}^{n} m\left(t_i\right)^2}}, \tag{32}$$

and

$$U2 = \sqrt{\frac{\sum_{i=1}^{n-1}\left(\frac{m(t_{i+1}) - m_{t_{i+1}}}{m_t}\right)^2}{\sum_{i=1}^{n-1}\left(\frac{m_{t_{i+1}} - m_{t_i}}{m_{t_i}}\right)^2}}. \tag{33}$$

Theil's U Statistics is a performance evaluation measure for the selected model, with U1 measuring the discrepancy between estimated results and actual data, ranging from 0 to 1. A lower U1 value indicates a more precise fit. U2 determines whether the selected model's results outperform naive forecasts that rely on historical averages. U2 also ranges from 0 to 1, and a value less than 1 implies superior performance compared to the forecasts.

(4) The Akaike information criterion (AIC) is defined as [8]:

$$AIC = 2k - 2\ln\left(L\right), \tag{34}$$

where k is the number of estimated parameters in the selected model and L is the likelihood function of the selected model at its maximum value. The smaller the value of AIC the better the fit. Since increasing the number of parameters of the model tends to yield a better fit, AIC also takes the degree of freedom into account.

(5) The *Prediction at levell (Pred* (l)*)* is defined as [32]:

$$Pred\ (l) = \frac{p}{n}, \tag{35}$$

where p indicates the number of estimated values that are within l of the actual value. For example, there is a model whose $Pred\,(0.25) = 80\%$, which indicates 80% of the prediction of the model falls within 25% of the actual values. Generally l is set as 0.25, and besides, $Pred\ (0.25) \geq 0.75$ is an acceptable performance criterion for a model [32]. The higher the value of $Pred(l)$ on a fixed l, the more precise the estimation of the model.

C. PERFORMANCE ANALYSIS

In this study, the Goel-Okumoto (GO) model, the Yamada delayed S-shaped (YDS) model, the Inflected S-shaped (IS) model, the Logistic growth (LG) model, the Gompertz growth (GG) model, and the original Diffusion (DF) model are considered as the candidate models for comparison [8], [25], [26]. All selected candidate models are listed in Table 2. To obtain estimates for all parameters of the candidate models, we employed the methods of least squares estimation (LSE) and maximum likelihood estimation (MLE) [34].

TABLE 2. Summary of candidate models.

Models	MVFs
The GO model	$m(t) = a(1-e^{-pt})$
The YDS model	$m(t) = a(1-(1+pt)e^{-pt})$
The IS model	$m(t) = \frac{1-e^{-pt}}{1+\psi e^{-pt}}$
The LG model	$m(t) = \frac{a}{1-ce^{-pt}}$
The GG model	$m(t) = ap^{c^t}$
The DF model	Equation (8)
Proposed model #1	Equation (16)
Proposed model #2	Equation (20)
Proposed model #3	Equation (25)
Proposed model #4	Equation (28)

1) DS1

First, Table 3 presents the estimated parameter values for all candidate models using LSE and MLE. As shown in Table 3, the additional detection rate values were not zero. Fig. 2 gives the plot of cumulative curves of actual and estimated number of faults for DS1. Table 4 shows the comparison results. Based on the results in Table 4, the DF outperforms the GO model in all criteria, indicating the presence of additional debugging phenomenon in DS1. Upon considering the delayed time lag, it was observed that the proposed Models #1 and #2 outperform the original diffusion model, indicating the presence of delayed time lag between detection and correction in DS1. Furthermore, the proposed Models #3 and #4 perform better than those that do not consider imperfect debugging when considering delayed time lag as well, suggesting the presence of imperfect debugging in the DS1. On the other hand, using either LSE or MLE as the estimation method, we can see that the all criteria value of the proposed Models #3 and #4 are ranked first or second. This means that the prediction of our proposed models is more accurate than other candidate models. Fig. 3 shows the Pred(l) plots of all candidate models at different levels. If one model has better performance in terms of Pred(l), its Pred(l) curve would lie on top of that for the other models. Similarly, Fig. 3 indicates the proposed Models #3 and #4 perform better at different levels. The performance of the GO model was the worst in the candidate models, since it was an exponential model.

2) DS2

For DS2, Table 5 presents the estimated parameter values for each candidate model. The plot of cumulative curves of actual and estimated number of faults for DS2 is illustrated graphically in Fig. 4. The actual curve of faults corrected in this dataset was found to be exponential, which favors the use of exponential SRGMs. The parameter q values in Table 5 are close to zero, indicating the absence of the phenomenon of additional debugging. Two primary cases may contribute to this outcome: inefficient testing procedures that fail to identify similar or related faults in a failure and a coding

TABLE 3. Results of parameter estimation for DS1.

(a) Using LSE

Models	*a*	*p*	*q*	*r*	*Ψ*	*c*
The GO model	1304.34	0.0422				
The YDS model	1007.03	0.144				
The IS model	947.59	0.204			8.10	
The LG model	928.55	0.255				18.24
The GG model	956.75	0.0101				0.845
The DF model	947.54	0.0223	0.181			
Proposed model #1	963.99	0.134	0.0827			
Proposed model #2	955.76	0.0323	0.165		161.67	
Proposed model #3	752.97	0.131	0.333	0.0172		
Proposed model #4	666.741	0.0245	0.292	0.0148	0.0214	

(b) Using MLE

Models	*a*	*p*	*q*	*r*	*Ψ*	*c*
The GO model	1175.09	0.0490				
The YDS model	1017.09	0.140				
The IS model	997.11	0.165			5.39	
The LG model	983.04	0.271				22.24
The GG model	1016.47	0.0174				0.862
The DF model	995.88	0.0268	0.140			
Proposed model #1	1010.67	0.137	0.0182			
Proposed model #2	969.70	0.0306	0.143		172.87	
Proposed model #3	751.04	0.133	0.281	0.0145		
Proposed model #4	651.18	0.0185	0.314	0.0149	0.00668	

style lacking in fault similarity and dependency. Based on the parameter values, it was challenging to determine which case applied to DS2. As mentioned in Section III-B, this dataset demonstrates that when q approaches zero, the diffusion model will simplify to the GO model.

Table 6 displays the comparison results for all selected models. Given that the proposed Model #2 outperforms the DF model, we can infer that the delayed time lag was indeed present in DS2. Additionally, the proposed Model #2 outperforms the proposed Model #1 in most criteria, while the proposed Model #4 outperforms the proposed Model #3 in most criteria. Therefore, we can conclude that DF2 is more suitable for describing the delayed time lag in DS2 than DF1. The parameter *r* values indicate that the phenomenon of imperfect debugging exists in DS2.

Furthermore, the proposed Model #3 outperforms the proposed Model #1, and the proposed Model #4 outperforms the proposed Model #2, providing further evidence of the existence of imperfect debugging in DS2. Regardless of the estimation method used, the proposed Models #2, #3, and #4 consistently rank among the top three in most criteria, demonstrating their flexibility as they can reduce to exponential models when the a ctual data shape is exponential. In addition to our models, the GO model performs better than the other candidate models, confirming the suitability of this dataset for exponential SRGMs. Fig. 5 displays the plots of all candidate models at various levels, indicating that the proposed Models #2, #3, #4, and the GO model perform better at different levels.

V. OPTIMAL RELEASE TIME AND MANAGEMENT

Determining the ideal release time is a crucial issue for software developers, often referred to as the optimal release-time problem. Over the years, several models and methods have been proposed to address this challenge [35], [36], [37], [38], [39], [40], [41], [42]. For example, Yang et al. [41] proposed a novel approach for modeling the reliability of multi-release open source software (OSS) using general masked data. Unlike traditional methods that use change point models, their proposed approach is based on an additive model that can accommodate general masked data.

Given that open-source software development typically involves frequent and early releases, it is essential to consider the potential advantages and drawbacks of early releases. As a result, OSS project developers must determine the optimal time to release their software, considering that issues are often

TABLE 4. Results of the model comparisons for DS1.

(a) Using LSE

Models	MSE	R^2	TS		Pred(0.25)	AIC
			U1	U2		
The GO model	4083.81	0.962	0.0432	5.534	0.784	
The YDS model	673.66	0.994	0.0175	1.278	0.865	
The IS model	800.05	0.993	0.0188	1.935	0.892	
The LG model	1407.44	0.987	0.0250	3.328	0.892	
The GG model	487.60	0.996	0.0165	1.030	0.919	
The DF model	800.05	0.993	0.0188	1.931	0.892	
Proposed model #1	521.42	0.995	0.0152	0.846	0.946	
Proposed model #2	650.455	0.994	0.0167	1.068	0.892	
Proposed model #3	237.32	0.998	0.0101	0.284	0.946	
Proposed model #4	227.67	0.998	0.00974	0.307	0.946	

(b) Using MLE

Models	MSE	R^2	TS		Pred(0.25)	AIC
			U1	U2		
The GO model	4617.93	0.957	0.0466	5.793	0.784	626.64
The YDS model	728.06	0.993	0.0183	1.142	0.865	414.44
The IS model	1184.23	0.989	0.0228	2.438	0.838	473.93
The LG model	1596.58	0.986	0.0264	2.730	0.892	690.98
The GG model	1262.16	0.989	0.0253	1.885	0.892	419.23
The DF model	1168.25	0.989	0.0227	2.431	0.838	473.93
Proposed model #1	710.51	0.994	0.0178	1.019	0.892	415.58
Proposed model #2	1477.63	0.987	0.0255	0.929	0.919	473.60
Proposed model #3	256.15	0.998	0.0105	0.378	0.946	380.68
Proposed model #4	319.95	0.997	0.0116	0.471	0.973	381.52

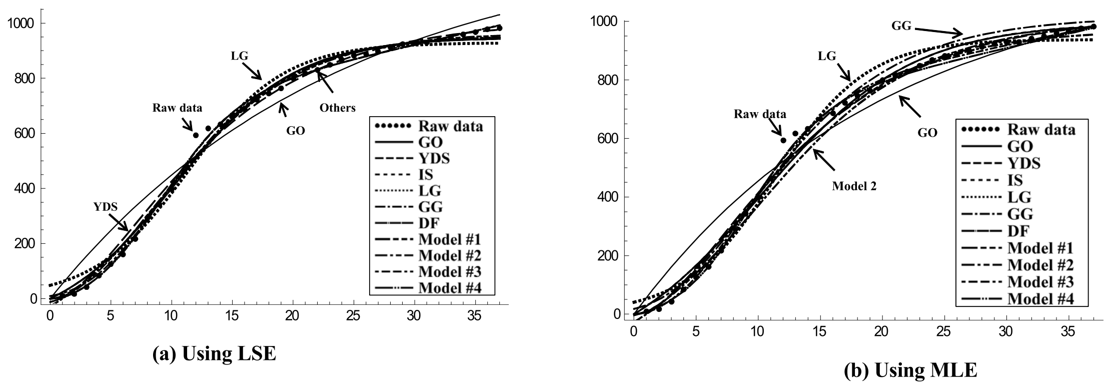
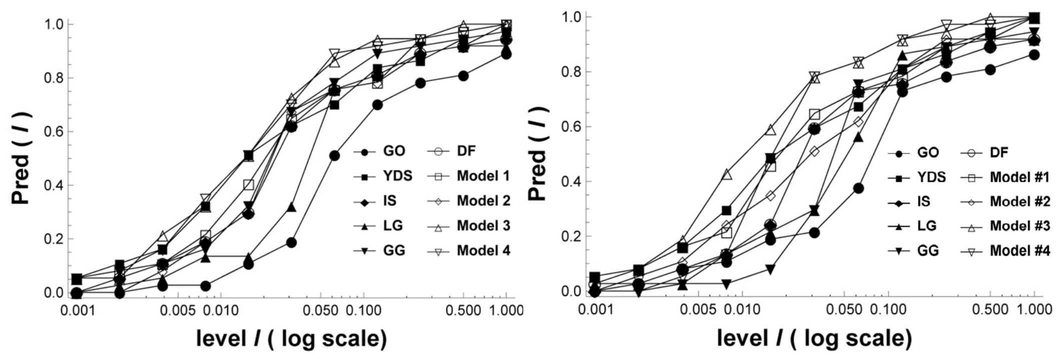


FIGURE 2. Cumuliative number of faults for DS1 and the fitted curves for all models.

addressed in subsequent versions. This section aims to examine various approaches and factors that software developers must consider when determining the optimal release time for their projects.

A. COST FACTORS

The calculation of the anticipated cost of open-source software (OSS) $EC(t)$ encompasses three factors: (i) the cost of testing, (ii) the cost incurred due to faults detected during



(a) Using LSE (b) Using MLE

FIGURE 3. Pred(l) for different l level (DS1).

TABLE 5. Results of parameter estimation for DS2.

(a) Using LSE

Models	*a*	*p*	*q*	*r*	*Ψ*	*c*
The GO model	842.77	0.0292				
The YDS model	703.25	0.0881				
The IS model	843.26	0.0292			4.21×10^{-11}	
The LG model	709.71	0.0881				6.01
The GG model	739.37	0.0929				0.942
The DF model	843.26	0.0292	1.03×10^{-12}			
Proposed model #1	703.26	0.0881	5.86×10^{-13}			
Proposed model #2	843.24	0.0302	6.33×10^{-11}		631.959	
Proposed model #3	439.29	0.152	7.69×10^{-6}	0.0152		
Proposed model #4	437.991	0.0463	0.0567	0.0119	9991.45	

(b) Using MLE

Models	*a*	*p*	*q*	*r*	*Ψ*	*c*
The GO model	894.99	0.0260				
The YDS model	780.37	0.0734				
The IS model	894.99	0.0260			4.08×10^{-8}	
The LG model	765.41	0.0826				7.23
The GG model	783.57	0.0894				0.945
The DF model	894.99	0.0260	8.05×10^{-10}			
Proposed model #1	779.87	0.0735	5.33×10^{-9}			
Proposed model #2	854.32	0.0300	6.30×10^{-15}		700.54	
Proposed model #3	438.56	0.155	2.08×10^{-10}	0.0148		
Proposed model #4	431.45	0.0451	0.0121	0.0121	0.0069	

operation, and (iii) the benefit or penalty of releasing the software earlier or later, respectively.

1) THE TESTING COST

The testing cost is computed by

$$C_1 \times t, \tag{36}$$

where C_1 is the cost per unit time for testing.

2) THE DAMAGE COST

The damage cost for fault detected during operation is given by

$$C_2\left(a - m\left(t\right)\right), \tag{37}$$

where C_2 is the damage cost per fault detected during operation.

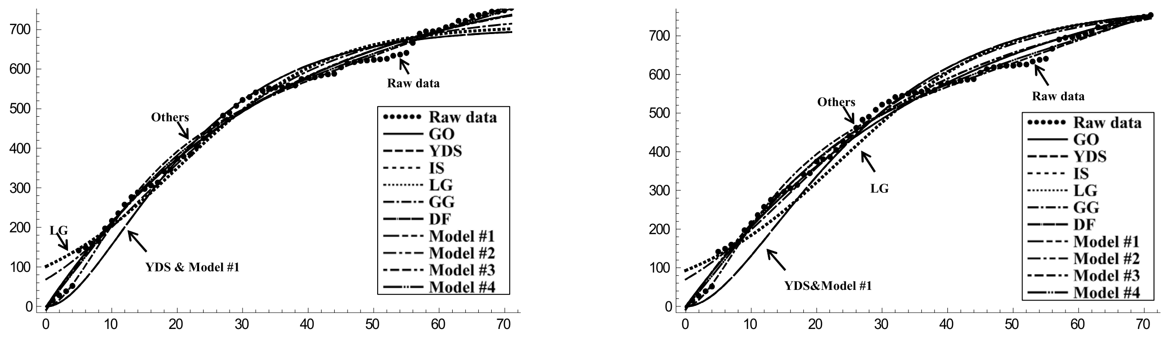
TABLE 6. Results of the model comparisons for DS1.

(a) Using LSE

Models	MSE	R^2	TS		Pred(0.25)	AIC
			U1	U2		
The GO model	297.79	0.993	0.0160	1.0436	0.944	
The YDS model	1332.53	0.969	0.0338	1.244	0.859	
The IS model	302.18	0.993	0.0160	1.0424	0.944	
The LG model	1177.35	0.973	0.0315	3.487	0.944	
The GG model	695.75	0.984	0.0214	2.446	0.944	
The DF model	302.18	0.993	0.0160	1.0436	0.944	
Proposed model #1	1352.12	0.969	0.0338	1.244	0.859	
Proposed model #2	300.98	0.993	0.0158	0.827	0.958	
Proposed model #3	318.64	0.993	0.0163	0.751	0.944	
Proposed model #4	228.81	0.995	0.0137	0.802	0.930	

(b) Using MLE

Models	MSE	R^2	TS		Pred(0.25)	AIC
			U1	U2		
The GO model	370.60	0.991	0.0178	0.953	0.944	752.72
The YDS model	2286.93	0.947	0.0436	1.462	0.803	899.56
The IS model	376.05	0.991	0.0178	0.953	0.944	754.72
The LG model	1905.53	0.957	0.0397	3.091	0.944	967.51
The GG model	1044.06	0.976	0.0254	2.479	0.944	838.26
The DF model	376.05	0.991	0.0178	0.953	0.944	754.72
Proposed model #1	2308.22	0.947	0.0435	1.463	0.803	901.56
Proposed model #2	342.74	0.992	0.0168	0.861	0.958	628.95
Proposed model #3	336.90	0.992	0.0168	0.726	0.944	760.69
Proposed model #4	234.42	0.995	0.0139	0.701	0.944	745.12



(a) Using LSE **(a) Using MLE**

FIGURE 4. Cumuliative number of faults for DS2 and the fitted curves for all models.

3) THE BENEFIT AND PENALTY FUNCTION

Due to the nature of open-source software development, early and frequent releases are common. This approach allows OSS projects to receive more user feedback and gain a larger market share than competitors who release their products later. Additionally, releasing OSS before the deadline can free up testing resources early, allowing developers to focus on other tasks. Therefore, if an OSS project can be released

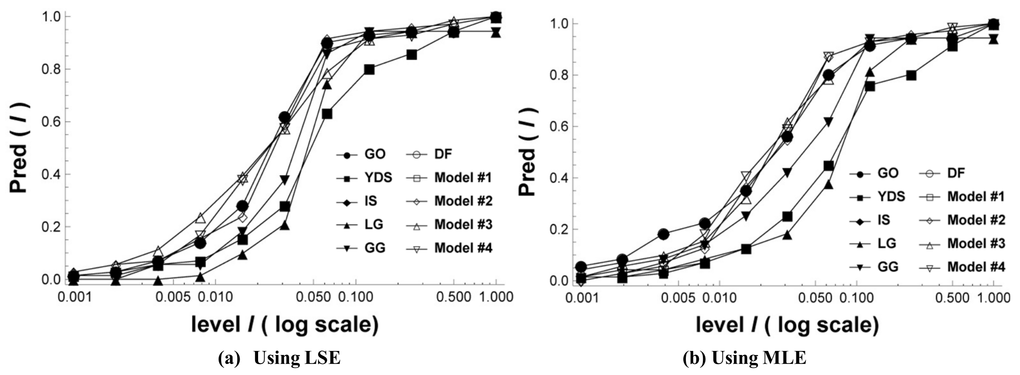


FIGURE 5. Pred(l) for different l level (DS2).

ahead of schedule, it can potentially benefit from increased market share, enhanced user feedback, and more time to work on other tasks. Here we propose to use the linear function to describe the benefit and the benefit function $B(t)$ as:

$$B(t) = \begin{cases} (D-t)B_1, & 0 \le t < D \\ 0, & t \ge D, \end{cases} \tag{38}$$

where D is the deadline for the software release and B_1 is the comprehensive benefit per unit of time for an early OSS release. In contrast, if an OSS project is released after the deadline D, in addition to the testing cost, it incurs a penalty cost due to the loss of market share, user feedback, and the delay of other work. This penalty cost is justifiable because, due to user familiarity, later releases of OSS may lose market share and user reactions, even if their reliability is higher than other OSS projects released earlier. As additional costs tend to increase rapidly in the early stages and gradually in the later stages, an exponential function is typically used to describe the penalty cost. Therefore, we can represent the penalty cost as the penalty function $P(t)$.

$$P(t) = \begin{cases} 0, & 0 \le t < D \\ P_1\left(1 - e^{-(t-D)}\right), & t \ge D, \end{cases} \tag{39}$$

where P_1 is the maximum penalty cost for releasing the OSS late.

Hence, the total expected cost $EC(t)$is given by:

$$EC(t) = \begin{cases} C_1 \cdot t + C_2(a - m(t)) - B(t), & 0 \le t < D \\ C_1 \cdot t + C_2(a - m(t)) + P(t), & t \ge D. \end{cases} \tag{40}$$

In order to derive a set of simple decision rules, we can rewrite (40) as:

$$EC(t) = \theta_1(t) I_{[0,D)}(t) + \theta_2(t) I_{[D,\infty)}(t), \tag{41}$$

where

$$\theta_1(t) = C_1 \cdot t + C_2(a - m(t)) - B(t), 0 \le t < \infty \tag{42}$$

$$\theta_2(t) = C_1 \cdot t + C_2(a - m(t)) + P(t), 0 \le t < \infty \tag{43}$$

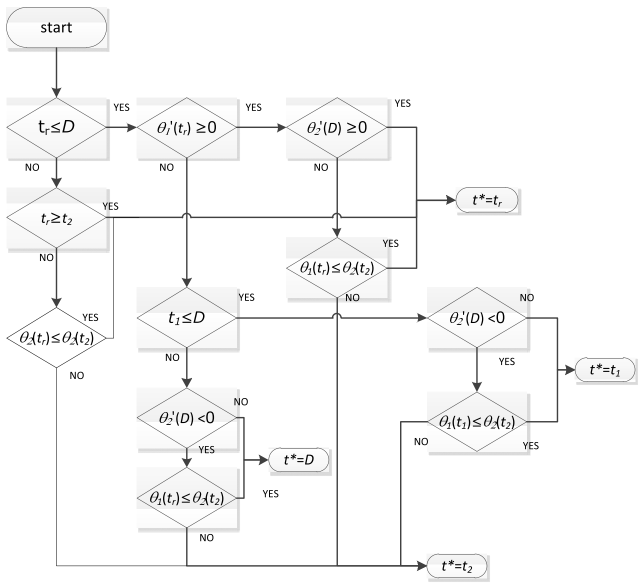


FIGURE 6. Decision flow chart.

and the indicator function I_A is:

$$I_A(x) = \begin{cases} 0, & t \notin A \\ 1, & t \in A. \end{cases} \tag{44}$$

Let t_1 and t_2 be the solutions of $\theta_1'(t_1) = 0$ and $\theta_2'(t_2) = 0$, respectively. OSS usually has an expected lower boundary of reliability LR. So we also have to know the lower boundary of release time t_r which can be obtained by the following:

$$LR = \frac{a}{m(t)}. \tag{45}$$

B. DECISION PROCEDURES

Although the OSS projects tend to be released early and often, the OSS should achieve the expected lower boundary of reliability before releasing. Hence, in consideration of the lower boundary of reliability, even if the cost function is at a minimum at time t, which is smaller than t_r, the OSS projects would also not release their products at time t. It is obvious

TABLE 7. Decision table.

t^*	$t_r \le D$			$t_r > D$
	$t_1 \ge t_r$		$t_1 < t_r$	
	$t_1 \le D$	$t_1 > D$		
t_r			$t_2 \le D$, or $t_2 > D$, $\theta_1(t_r) \le \theta_2(t_2)$	$t_2 \le t_r,\ \theta_2(t_r) \le \theta_2(t_2)$
t_1	$t_2 \le D$, or $t_2 > D$, $\theta_1(t_1) \le \theta_2(t_2)$			
D		$t_2 \le D$, or $t_2 > D$, $\theta_1(D) \le \theta_2(t_2)$		
t_2	$t_2 > D,\ \theta_1(t_1) > \theta_2(t_2)$	$t_2 > D,\ \theta_1(D) > \theta_2(t_2)$	$t_2 > D,\ \theta_1(D) > \theta_2(t_2)$	$t_2 > t_r,\ \theta_2(t_r) > \theta_2(t_2)$

TABLE 8. Data and results for examples 1-4.

Parameters	$t_r \le D$			$t_r > D$
	Example 1	Example 2	Example 3	Example 4
C_1	9,000	10,000	20,000	20,000
C_2	6,000	3,500	6,000	6,000
B_1	12,500	4,500	1,000	5,000
P_1	10,000	10,000	0	10,000
D	7	7	12	7
LR	0.7	0.7	0.5	0.95
p	0.5	0.3	0.1	0.2
q	0.6	0.3	0.4	0.3
a	50	50	50	50
t_r	3.77	6.50	11.01	14.59
t_1	5.45	$>D$	$>D$	—
t_2	$<D$	7.64	13.16	$< t_r$
$\theta_1'(t_r)$	<0	<0	<0	—
$\theta_2'(D)$	>0	<0	<0	—
$\theta_1(t_r)$	83,486	114,932	369,116	—
$\theta_1(t_1)$	57,833*	—	—	—
$\theta_1(D)$	71,056	114,085*	366,817	—
$\theta_2(t_r)$	—	—	—	316,823*
$\theta_2(t_2)$	—	116,001	364,226*	—
t^*	t_1	D	t_2	t_r
$m(t^*)/a$	0.91	0.75	0.98	0.95

that the optimal time in all cases are a point of D, t_r, t_1 and t_2. Then, by arranging all potential cases, we can summarize the necessary conditions for the optimal time decision, which is provided in Table 7. Moreover, from Table 7, we further derive the optimal time decision procedure in Fig. 6 where t^* is the optimal release time. We strongly believe that this is a valuable and convenient tool for OSS developers to ascertain the optimal timing for the release of their products.

C. EXAMPLES

To demonstrate the decision-making process for determining the optimal release time, we have selected four examples. The corresponding parameter settings and the required calculations for all these examples can be found in Table 8.

1) EXAMPLE 1

In this example, the benefit for early release is relatively high and the LR is medium. This implies that the target market for this OSS project prefers to receive the product early on in its development cycle, with a moderate level of reliability expected at the time of release. We determine that the optimal release t^* is always t_1 by using Table 7 and Fig. 6. The total cost we calculate with the OSS being released at t_1, is minimal. After the OSS is released at t_1, the proportion of removed faults in the fault content is 91%. This is higher than LR.

2) EXAMPLE 2

In this example, the end users are willing to tolerate encountering failures when using the OSS, as they see the benefits of an early release coupled with a moderate level of reliability, and the cost associated with a fault detected during operation is relatively low. By our decision procedure, the optimal release time is D. After the OSS is released at D, the proportion of removed faults in the fault content is 75%. This is higher than LR.

3) EXAMPLE 3

In this case, the benefits of an early release for the OSS are relatively minimal, and there is no penalty for a later release. Therefore, the early release of the OSS would only result in marginal gains, as the target users place a greater emphasis on the quality and features of the OSS rather than its release time. Based on our decision procedure, we recommend that the optimal release time be set to t_2. The proportion of removed faults in the fault content is 98% after the OSS is released at t_2.

4) EXAMPLE 4

In this example, the OSS is assumed to be a life-critical system, which means that the requirements for reliability and the costs associated with testing are relatively high. As the target users expect the OSS to operate with a high level of reliability, the testing process needs to be more rigorous. Due to $t_r > D$ and $t_2 \leq t_r$, we can derive the optimal release time as t_r. After the OSS is released at t_r, the proportion of removed faults in the fault content is 95%, which is equal to LR.

VI. CONCLUSION

During software development, most developers have likely experienced situations where they choose to reuse existing code instead of writing a similar code from scratch. However, recent studies have shown that reusing replicated code can increase the likelihood of introducing faults. This is because developers may simply copy and paste the code without making necessary modifications. Therefore, it is important to acknowledge that when a fault is detected in a piece of replicated code, developers should carefully check other copies to ensure that similar faults are not present. By doing so, they can prevent the spread of potential errors throughout their codebase and ensure that their software remains reliable and maintainable. In the past, several software reliability growth models (SRGMs) were developed to analyze and predict the reliability of software systems. However, many of these models assume that developers can only detect one fault that causes a failure, which may not be the case in real-world scenarios.

Additionally, most traditional SRGMs are built on the same set of assumptions that may not hold true, particularly when considering the development of OSS. For instance, the traditional assumptions of perfect and immediate debugging do not align with the reality of OSS development. Debugging in OSS is often done collaboratively. This makes it difficult to achieve perfect and immediate debugging, which can impact the reliability of the software. To account for this, we use the diffusion model as our basic model and demonstrate that it is a special case of the delayed generalized NHHP model. Based on the delayed generalized NHHP model, we derived the extended diffusion models, which consider the phenomena of debugging time lag and imperfect debugging by modifying the assumptions. To validate our models, we used two real OSS projects as our datasets. The experimental results indicate that our proposed models can more accurately describe and predict the fault correction process compared to other models. Furthermore, our models show that the additional debugging effort, debugging time lag, and imperfect debugging are prevalent in most OSS projects. This provides valuable insights for developers in designing and implementing effective fault detection and correction strategies for OSS projects.

Finally, to apply the proposed models in determining the optimal release time for OSS, a decision procedure is proposed and discussed. By following the suggested decision procedure, developers can make a decision about when to release their software, taking into account the cost factors, the potential for faults and the needed reliability. This approach can help ensure that the software meets the needs of users while also maintaining its desired reliability.

REFERENCES

[1] J. D. Musa, A. Iannino, and K. Okumoto, *Software Reliability, Measurement, Prediction and Application*. New York, NY, USA: McGraw-Hill, 1987.

[2] M. R. Lyu, *Handbook of Software Reliability Engineering*. New York, NY, USA: McGraw-Hill, 1996.

[3] C.-Y. Huang, M. R. Lyu, and S.-Y. Kuo, "A unified scheme of some nonhomogenous Poisson process models for software reliability estimation," *IEEE Trans. Softw. Eng.*, vol. 29, no. 3, pp. 261–269, Mar. 2003.

[4] T. Kim, S. Park, and T. Lee, "Applying software reliability engineering process to software development in Korea defense industry," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops (ISSREW)*, Toulouse, France, Oct. 2017, p. 81.

[5] K. J. Chen, "Applying modified diffusion models to reliability assessment and management for open source software system," M.S. thesis, Inst. Inf. Syst. Appl., Nat. Tsinghua Univ., Hsinchu, Taiwan, 2014.

[6] E. S. Raymond, *The Cathedral and the Bazaar*. Sebastopol, CA, USA: O'Reilly, 1999.

[7] E. Capra, C. Francalanci, and F. Merlo, "An empirical study on the relationship between software design quality, development effort and governance in open source projects," *IEEE Trans. Softw. Eng.*, vol. 34, no. 6, pp. 765–782, Nov. 2008.

[8] F. M. Bass, "A new product growth model for consumer durables," *Manage. Sci.*, vol. 15, no. 5, pp. 215–227, 1969.

[9] P. K. Kapur, O. Singh, and R. Mittal, "Software reliability growth and innovation diffusion models: An interface," *Int. J. Rel., Quality Saf. Eng.*, vol. 11, no. 4, pp. 339–364, Dec. 2004.

[10] H. Pham, *System Software Reliability*. New York, NY, USA: Springer-Verlag, 2006.

[11] Z. Liu, S. Yang, M. Yang, and R. Kang, "Software belief reliability growth model based on uncertain differential equation," *IEEE Trans. Rel.*, vol. 71, no. 2, pp. 775–787, Jun. 2022.

[12] S. Yang, X. Gou, M. Yang, Q. Shao, C. Bian, M. Jiang, and Y. Qiao, "Software bug number prediction based on complex network theory and panel data model," *IEEE Trans. Rel.*, vol. 71, no. 1, pp. 162–177, Mar. 2022.

[13] P. Vizarreta, K. Trivedi, B. Helvik, P. Heegaard, A. Blenk, W. Kellerer, and C. Mas Machuca, "Assessing the maturity of SDN controllers with software reliability growth models," *IEEE Trans. Netw. Service Manage.*, vol. 15, no. 3, pp. 1090–1104, Sep. 2018.

[14] 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.

[15] Y. P. Wu, Q. P. Hu, M. Xie, and S. H. Ng, "Modeling and analysis of software fault detection and correction process by considering time dependency," *IEEE Trans. Rel.*, vol. 56, no. 4, pp. 629–642, Dec. 2007.

[16] N. F. Schneidewind, "An integrated failure detection and fault correction model," in *Proc. Int. Conf. Softw. Maintenance*, Montreal, QC, Canada, 2002, pp. 238–241.

[17] M. Xie and M. Zhao, "The schneidewind software reliability model revisited," in *Proc. 3rd Int. Symp. Softw. Rel. Eng.*, 1992, pp. 184–192.

[18] R. Xie, H. Qiu, Q. Zhai, and R. Peng, "A model of software fault detection and correction processes considering heterogeneous faults," *Qual. Rel. Eng. Int.*, pp. 1–17, Aug. 2022, doi: 10.1002/QRE.3172.

[19] 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.

[20] S. Yamada, K. Tokuno, and S. Osaki, "Imperfect debugging models with fault introduction rate for software reliability assessment," *Int. J. Syst. Sci.*, vol. 23, no. 12, pp. 2241–2252, Dec. 1992.

[21] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Trans. Softw. Eng.*, vol. 32, no. 3, pp. 176–192, Mar. 2006.

[22] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Proc. 2nd Work. Conf. Reverse Eng.*, 1995, p. 86.

[23] V. Mahajan and R. A. Peterson, *Models for Innovation Diffusion*. Thousand Oaks, CA, USA: SAGE, 1985.

[24] H. Pham, L. Nordmann, and Z. Zhang, "A general imperfect-software-debugging model with S-shaped fault-detection rate," *IEEE Trans. Rel.*, vol. 48, no. 2, pp. 169–175, Jun. 1999.

[25] 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.

[26] S. Yamada, M. Ohba, and S. Osaki, "S-shaped software reliability growth models and their applications," *IEEE Trans. Rel.*, vol. R-33, no. 4, pp. 289–292, Oct. 1984.

[27] *The Eclipse Bugzilla*. Accessed: Mar. 13, 2023. [Online]. Available: https://bugs.eclipse.org/bugs/

[28] *Wikipedia: The Free Encyclopedia*, Eclipse (Software), Wikimedia Foundation, San Francisco, CA, USA, Accessed: Mar. 13, 2023.

[29] *Wikipedia: The Free Encyclopedia*, Apache Tomcat, Wikimedia Foundation, San Francisco, CA, USA, May 2014.

[30] G. Keller and B. Warrack, *Statistics for Management and Economics*. Plymouth, MA, USA: Duxbury, 1999.

[31] H. Theil, *Applied Economic Forecasting*. Amsterdam, The Netherlands: North Holland, 1966.

[32] J. Lin and C. Huang, "Queueing-based simulation for software reliability analysis," *IEEE Access*, vol. 10, pp. 107729–107747, 2022.

[33] S. D. Conte, H. E. Dunsmore, and V. Y. Shen, *Software Engineering Metrics and Models*. Redwood City, CA, USA: Benjamin-Cummings, 1986.

[34] Å. Björck, *Numerical Methods for Least Squares Problems*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1996, doi: 10.1137/1.9781611971484.

[35] Y. F. Li, M. Xie, and T. N. Goh, "A study of the non-linear adjustment for analogy based software cost estimation," *Empirical Softw. Eng.*, vol. 14, no. 6, pp. 603–643, Dec. 2009.

[36] H. S. Koch and P. Kubat, "Optimal release time of computer software," *IEEE Trans. Softw. Eng.*, vols. SE–9, no. 3, pp. 323–327, May 1983.

[37] V. Nagaraju and L. Fiondella, "Online optimal release time for non-homogeneous Poisson process software reliability growth model," in *Proc. Annu. Rel. Maintainability Symp. (RAMS)*, Palm Springs, CA, USA, Jan. 2020, pp. 1–6.

[38] P. Prashant, A. Tickoo, S. Sharma, and J. Jamil, "Optimization of cost to calculate the release time in software reliability using Python," in *Proc. 9th Int. Conf. Cloud Comput., Data Sci. Eng.*, Noida, India, 2019, pp. 471–474.

[39] S. Chatterjee, B. Chaudhuri, C. Bhar, and A. Shukla, "Modeling and analysis of reliability and optimal release policy of software with testing domain coverage efficiency," in *Proc. 5th Int. Conf. Rel., INFOCOM Technol. Optim. (Trends Future Directions) (ICRITO)*, Noida, India, Sep. 2016, pp. 90–95.

[40] V. Kumar, V. B. Singh, A. Dhamija, and S. Srivastav, "Cost-reliability-optimal release time of software with patching considered," *Int. J. Rel., Quality Saf. Eng.*, vol. 25, no. 4, Aug. 2018, Art. no. 1850018.

[41] J. Yang, J. Chen, and X. Wang, "EM algorithm for estimating reliability of multi-release open source software based on general masked data," *IEEE Access*, vol. 9, pp. 18890–18903, 2021.

[42] A. Anand, M. Agarwal, Y. Tamura, and S. Yamada, "Economic impact of software patching and optimal release scheduling," *Quality Rel. Eng. Int.*, vol. 33, no. 1, pp. 149–157, Feb. 2017.

KUAN-JU CHEN received the B.E. degree in business administration from the National Kaohsiung University of Science and Technology, Kaohsiung, Taiwan, in 2012, and the M.S. degree in information systems and applications from National Tsing Hua University, Hsinchu, Taiwan, in 2014. He is currently a Software Development Team Leader with Garmin, Taiwan. His research interests include software testing and software reliability.

CHIN-YU HUANG (Member, IEEE) received the M.S. and Ph.D. degrees in electrical engineering from National Taiwan University, Taipei, in 1994 and 2000, respectively. He was with the Bank of Taiwan, from 1994 to 1999, and a Senior Software Engineer with Taiwan Semiconductor Manufacturing Company, from 1999 to 2000. 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. Before joining NTHU, in 2003, he was a Division Chief of the Central Bank of China, Taipei. 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 editorial board of *Journal of Information Science and Engineering*, since 2016. He is also serving as an Associate Editor for the IEEE Transactions on Reliability.