Software Reliability Modeling and Methods: A State of the Art Review

Mengmeng Zhu and Hoang Pham

Abstract The increasing dependence of our modern society on software systems has driven the development of software products to be more competitive and time-consuming. At the same time, large-scale software development is still considered a complex, effort-consuming, and expensive activity, given the influence of the transitions in software development, which are the adoption of software product lines, software development globalization, and the adoption of software ecosystems. Hence, the consequences of software failures become costly and even dangerous. In this chapter, we thus review probabilistic software reliability models with different groups. Since nonhomogeneous Poisson process (NHPP) based software reliability models have been successful tools in practical software reliability engineering, this chapter mainly focuses on the review of NHPP based software reliability models that address various concerns in software development practices, such as testing efficiency, testing coverage, multiple fault types, time-delay fault removal, and environmental factors.

Keywords Nonhomogenous Poisson process · Software reliability growth model · State of the art review

1 Introduction

In today's technological world, almost everyone is directly or indirectly in contact with computer software. Computers have been rapidly expanding to a wide array of complex machinery and equipment applied in our everyday safety, security, infrastructure, transportation system, and financial management. Since software product

M. Zhu (✉)
Department of Textile Engineering, Chemistry and Science, North Carolina State University, Raleigh, NC 27606, USA
e-mail: mzhu7@ncsu.edu

H. Pham
Department of Industrial and Systems Engineering, Rutgers University, Piscataway, NJ 08854, USA
e-mail: hopham@rci.rutgers.edu

A. G. Aggarwal et al. (eds.), *Optimization Models in Software Reliability*,
Springer Series in Reliability Engineering,
https://doi.org/10.1007/978-3-030-78919-0_1

is extensively involved in various industries and service-based applications, the increasing dependence of our modern society on software-driven systems has led the development of software product to be very competitive and time-consuming (Febrero et al. 2016). Unlike hardware systems, software cannot break down or wear out during its life cycle but can fail or malfunction under certain configurations within specific conditions (Hartz et al. 1997). Hence, the development, measurement, and qualifying of software are challenging yet critical in such a fast-growing technological society.

In 1980, Lehman (1980) summarized the *Laws of Program Evolution.* The first law, *Continuing Change*, expressed the universally observed fact that large programs are never completed. They just continue to evolve until the more cost-effective updated version replaces the systems. The second law, *Increasing Complexity*, could also be viewed as an instance of the second law of thermodynamics. As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases as well, unless the mission is done, or maintenance is needed. The third law, *The Fundamental Law of Program Evolution*, is subject to dynamics that make the programming process, measures system attributes and collaborative projects, and self-regulating with statistically proven trends and invariances. The fourth law, *Conservation of Organizational Stability (Invariant Work Rate)*, and the fifth law, *Conservation of familiarity*, both lead to the third law. The fourth law more focuses on the steadiness of multiloop self-stabilizing systems. A well-established organization is good at avoiding dramatic change and particularly discontinuities in the increasing growth of an organization. Especially in the past two decades, the complexity of the task that the software system performs has grown dramatically, faster than hardware due to the fast-paced high technology development (Catelani et al. 2011).

A modern software product is prone to include a large number of modules, system components, and Lines of Code (LOC) (Catelani et al. 2011; Han et al. 2012; Chang et al. 2014; Chatterjee et al. 1997; Chatterjee and Singh 2014). The size of a software product is no longer measured in terms of thousands of LOC, but millions of LOC. The latest investigation states that more than 10 Microsoft commercial software products could have more than 600 million LOC (Dang et al. 2011). In view of such a great amount of LOC, the complexity of software product, domain knowledge of programmer/tester, testing methodologies, testing coverage, and testing environment should be all carefully taken into account in software development.

Since the inception of electronic computing in the late 1940s, the development race of the computer industry has led to an unprecedented process (Patterson and Hennessy 2013). Powerful, inexpensive computer workstations replaced the drafting boards of circuits and computer designers. Moreover, an increasing number of design steps were automated. Computer and communication industries have grown into the largest, amongst the new rising industries in the twentieth century (Moravec 1998). Hardware advances have allowed software programmers to create wonderful coding and develop new features and functionality (Moravec 1998). However, there exists uneven progress between software and hardware in the computer revolution in the past few decades. Based on the latest technology review, hardware is leaving software behind. As a matter of fact, software is relied on a less firm foundation, at the same time, carries a larger burden than hardware in operation. Given the current technology in manufacturing and electrical engineering, software has more potentials to allow designers to contemplate more ambitious systems in consideration of a broader multidisciplinary scope (Lyu 1996, 2007).

The nonperformance and failures of a software system are inconvenient, sometimes can lead to severe consequences, especially in the application of aerospace engineering and national defense systems. In March 2015 (COMPUTERWORLD 2020), a software glitch carried in the software package of Lockheed Martin F-35 Joint Strike Fighter aircraft had made the aircraft could not correctly detect the target. The sensor on the plane cannot distinguish the difference between singular and multiple threats. Additionally, different F-35 aircraft provide different detection information even they are aiming at the same threat, which depends on the angles they are aiming at and what their sensors have received. The delivery date had to be postponed as well because of this issue.

Software failures can also cause serious consequences in an automobile. Toyota had to recall almost 2 million Prius hybrid vehicles, in order to fix a software glitch along with its engine control units (ECUs) in February 2014 (COMPUTERWORLD 2020). A malfunction within the car's hybrid drive system caused by a software glitch could, in certain circumstances, cut the system's power and cause the car to an unscheduled halt. A software glitch affecting the ECUs controlling the motor/generator and the hybrid system could put extra thermal stress on certain transistors under certain conditions. The same software issue recurred in July 2015, which has resulted in the recall of 625,000 Prius cars globally. Software failures have affected the healthcare system as well. Emergency services were unavailable for around six hours across seven U.S. states in April 2014 (COMPUTERWORLD 2020). The incident had a major impact on 81 call centers, meaning about 6,000 people who made 911 calls that were not able to connect in these seven states. There is a study announced by the Federal Communications Commission found that the cause of service unavailable was an entirely preventable software error.

The nonperformance and failures of software are expensive. A study carried by the National Institute of Standards & Technology in 2002 found that inadequate infrastructures for fixing software bugs cost the U.S economy \$59.5 billion every year. What about the global cost of fixing software bugs every year? This study also estimated that more than a third of software bugs could be eliminated by improving software testing scheduling and methodology (Tassey 2002).

Hence, developing reliable software is a major challenge to the software industry, information technology (IT) industry, and other related industries, which leads to the fact that *Software Reliability Engineering* is popular in both academia and industry. We thus discuss the recent trends in software development and the importance of developing software reliability models in Sect. 2. Indeed, the deterministic and probabilistic software reliability models are reviewed in Sect. 3. Since nonhomogeneous Poisson process (NHPP) based software reliability models have been successful tools in practical software reliability engineering, Sect. 4 thus focuses on the review of NHPP based software reliability models that address various concerns in the software development practices. Section 5 concludes this chapter.

2 Software Reliability Engineering

2.1 Trends in Software Development

Large-scale software development is a very complex, effort-consuming, and expensive activity. Even though many innovations and improvements have been proposed on software architecting systems and development approaches, large-scale software development is still largely unpredictable and error-prone (Bosch and Bosch-Sijtsema 2010).

Bosch and Bosch-Sijtsema (2010) discussed three trends in software development in 2010, which will further accelerate the complexity of large-scale software development. The first trend is the increasing adoption of software product lines. A software product line consists of a software platform shared by a group of products. Each software product can select and configure components in the platform and extend the platform with desirable functionality. At the same time, the platform consists of many components with the associated team. Each team takes charge of one product or several products. Software development is taken place within many teams in the organization. During the development cycle, the interactions and communications among teams are much than traditional software development teams. Some research identified the adoption of software product lines allows 50–75% of development expenses reduction and decreases the defect density if the adoption is successful (Hallsteinsen et al. 2008; Clements and Northrop 2002). However, the adoption of software product lines also brings a new level of dependency on organizations in software development. The second trend is software development globalization. Companies often have multiple software development sites globally or partnered with other remote companies especially located in India and China. There are many advantages in terms of software development, e.g., cycle time reduction, travel cost reduction, fewer communication issues about user experiences, and faster response to customers (Cascio and Shurygailo 2003). Nevertheless, software development globalization also brings challenges given the culture difference, time zone, software engineering maturity in every country, and technical skills between different countries. The third trend is the adoption of software ecosystems. A software ecosystem is defined as a set of businesses functioning as a unit and interacting with a shared market for software and services. There are relationships amongst those units, which are supported by a technological platform, operating through the exchange of information, resources, and artifacts (Messerschmitt and Szyperski 2005; Jansen et al. 2007). Software ecosystem also takes external developers, domain experts, and users. Hence, community-centric collaboration and coordination are very important, which are similar to the adoption of software product lines (the first trend in software development). Thus, the dependencies between components will increase and the complexity of software development will increase accordingly (Bosch and Bosch-Sijtsema 2010).

2.2 Importance of Software Reliability Model

Given the complexity of large-scale software development, how we can assure software quality is one of the challenging problems in the industry. One of the fundamental quality characteristics is reliability. It is generally accepted that reliability is the key factor in software quality since it quantifies failures and misbehaviors of the product. As recognized in both industry and academia, reliability is an essential measurement metric for developing a robust and high-quality software product (Febrero et al. 2016; Lyu 1996, 2007). According to the definition given by ANSI/IEEE, software reliability is defined as the probability that a software system can perform its designed function without failure during a specified time on a given set of inputs under defined environments (Lyu 1996).

On the other hand, the increasing complexity and shortened iteration cycle of software products bring in a decrease in average market life expectancy (Tassey 2002). Thus, since the 2000s, there is a great attention shift from hardware development and testing to improve software quality and reliability with the purpose of winning more market share. Moreover, high reliability is desirable if a software company plans to reduce the total cost of software products from an economic point of view. It is undoubtedly that lower reliability software product not only results in the negative impact regarding customer satisfaction but also brings in the additional cost occurred during the operation phase because fixing a software fault in the operation phase costs more resources compared with in-house testing. Since the fixing cost for a software fault in the operation phase is much higher than the in-house testing phase, most organizations try to minimize these expenditures occurred in the operation phase. That is why many high technology organizations need to release multiple versions for a software product instead of fixing software faults in the operation phase, to improve product reliability and introduce new features to improve the user experience.

It is necessary to develop a practical and applicable model that can capture the software failure growth trend, predict the number of failures and software reliability given a specific period of operating time, propose the optimal release time of new products, and schedule the delivery time for the next release based on the predetermined level of reliability. Thus, software reliability models are applied to evaluate software reliability and capture the failure growth trend in the past few decades. There are several ways to measure software reliability. A practical and common one is model software reliability by utilizing the past failure behaviors obtained from the testing phase.

3 Software Reliability Models

Software failure data, collected mostly in the testing phase, are applied to study the behavior of software systems, such as software reliability given a specific time interval, failure growth rate, the number of remaining faults in the system, and the optimal release time. Hence, a great number of studies have been focused on the development of software reliability models in the past four decades with various assumptions, such as testing methodology, testing coverage, fault removal efficiency, fault dependency, and time-delay debugging.

The classification of software reliability models was presented by different researchers (Bastani and Ramamoorthy 1986; Goel 1985; Musa et al. 1990; Mellor 1987). One of the widely utilized classification methodologies categorizes software reliability models into two types: the deterministic models and the probabilistic models (Pham 2000). The deterministic models are used to study: (1) the element of a program by counting the number of distinct operators, operands, errors, and instruction; (2) the control flow of a program by counting the branches; (3) the data flow of a program (data sharing and passing). There are two well-known models: Halstead's software metric (Halstead 1977) and McCabe's cyclomatic complexity metric (McCabe 1976). These models provided an innovative and pioneering quantitative approach to analyze and measure the performance of software systems at that time; however, a random event is not involved, hence, these models are not suitable to apply in modern software. The probabilistic models take into account failure detection and failure removal as probabilistic events during software development. The classification of the probabilistic software reliability models is given by Pham (2000, 2007), Xie (1991): (1) error seeding; (2) failure rate; (3) curve fitting; (4) reliability growth; (5) Markov structure; (6) time series; (7) NHPP.

In Sect. 3.1, we review the research articles regarding the probabilistic software reliability models with groups stated as follows: error seeding, failure rate, curving fitting, reliability growth, Markov structure, and time series. Since NHPP based software reliability models have been successful tools in practical software reliability engineering, this chapter mainly focuses on the review of NHPP based software reliability models. Therefore, Sect. 3.2 introduces the general theory of NHPP and Sect. 4 reviews a great number of NHPP software reliability models with different considerations, such as testing effort, testing coverage, fault removal efficiency, fault dependency, time-delay fault removal, environmental factor on affecting software reliability, and multiple-release software.

3.1 Probabilistic Software Reliability Models

In the error seeding based software reliability model, the number of errors in a program is estimated by applying the multi-stage sampling technique (Mills 1972; Cai 1998; Tohma et al. 1991). Errors are categorized as indigenous errors and induced (seeded) errors. The number of indigenous errors, which is unknown, is estimated from the number of induced errors and the ratio of these two types of errors obtained from software debugging data. We list three models in the group of errors seeding based software reliability models.

Mills (1972) proposed an error seeding method to estimate the number of errors in a program by introducing seeded errors into the program. If the probability of detecting both indigenous errors and induced errors are equal, then the probability of k induced errors in r removed errors follows a hypergeometric distribution, given as

$$P(k; N, n_1, r) = \frac{\binom{n_1}{k}\binom{N}{r-k}}{\binom{N+n_1}{r}}, \ k = 1, 2, \ldots, r$$

where N is the total number of indigenous errors, n_1 is the total number of induced errors, r is the total number of errors removed during debugging, k is the total number of induced errors in r removed errors, and $r - k$ is the total number of indigenous errors in r removed errors. Given the parameters n_1,r, and k are known, the maximum likelihood estimation (MLE) of N can be shown as (Huang 1984) $\widehat{N} = \lfloor N_0 \rfloor + 1$, in which $N_0 = n_1(r-k)/k - 1$. Note that N_0 and $N_0 + 1$ are both MLEs of N if N_0 is an integer. Tohma et al. (1991) introduced a reliability model based on the hypergeometric distribution to estimate the number of errors in the program. Later, Cai (1998) modified Mills' model by dividing software into two parts, Part 0 and Part 1.

In the failure rate class, these studies (Jelinski and Moranda 1972; Schick and Wolverton 1978; Moranda 1981) focused on how failure rates change at the failure time during the failure intervals. The number of faults in the program is a discrete function, thus, the failure rate of a program is a discrete function as well. The Jelinski-Moranda model (Jelinski and Moranda 1972) is one of the earliest software reliability models, which states the program failure rate at the i^{th} failure interval is given by $\lambda(t_i) = \emptyset[N - (i-1)], \ i = 1, \ 2, \ldots, \ N$, in which $\emptyset$ is a proportional constant representing the contribution of one fault makes to the overall program, and N is the number of initial faults in the program. The Schick-Wolverton model (Schick and Wolverton 1978) modified the Jelinski-Moranda model by assuming the failure rate at the i^{th} time interval increases with time t_i since the last debugging. Later, Moranda (1981) proposed a reliability model considering the program failure rate function as initially a constant D and decreases geometrically at failure times.

In the curve fitting class (Belady and Lehman 1976; Miller and Sofer 1985), the models use statistical regression analysis to illustrate the relationship amongst software complexity, the number of faults, and failure rate in the software. Linear regression analysis, nonlinear regression analysis, or time series approach is applied between the dependent and independent variables. Estimation of errors, complexity, and failure rate are investigated in the modeling. Belady and Lehman (1976) introduced a model by applying time series approach to estimate software complexity. Miller and Sofer (1985) also proposed a model to estimate software failure rate by assuming the failure rate is a monotonically non-increasing function.

In the reliability growth class (Coutinho 1973; Wall and Ferguson 1977), the improvement of program reliability is measured and predicted via the testing phase by reliability growth models. The failure rate is a function of time or the number of testing cases in this group of models. Coutinho (1973) pointed out that the failure rate is a function of the cumulative number of failures and testing time. Wall and Ferguson (1977) proposed a model that is similar to Weibull model to predict software failure rate during testing.

In the group of Markov structure models (Goel and Okumoto 1979a; Littlewood 1979; Yamada et al. 1998; Goseva-Popstojanova and Trivedi 1999; Dai et al. 2005), the assumption is that the failure of the modules is independent of each other. Goel and Okumoto (1979a) proposed a linear Markov model with imperfect debugging. Meanwhile, they gave the transition probability of the model. Littlewoods (1979) developed a reliability model incorporating the transitions between modules while operating. Two types of failures are considered: failure from each module, modeled as a Poisson failure process, and failure from the interface between modules. Yamada et al. (1998) performed a software safety model to illustrate software's time-dependent behavior using the Markov process. Goseva-Popstojanova and Trivedi (1999) proposed a software reliability modeling framework that can consider the phenomena of failure correlation and further studied its effects on the software reliability measures based on the Markov renewal process. Dai et al. (2005) proposed a software reliability model based on a Markov renewal process for the modeling of the dependence among successive software runs, in which four types of failures are allowed in the general formulation. Meanwhile, the cases of restarting with repair and without repair are considered.

In the time series model group (Chatterjee et al. 1997; Singpurwalla and Soyer 1985; Ho and Xie 1998; Xie and Ho 1999), autoregressive integrated moving average method is applied to study software reliability. Singpurwalla and Soyer (1985) introduced several ramifications into a random coefficient autoregressive process of order 1 to describe software reliability. Besides, several research papers (Chatterjee et al. 1997; Ho and Xie 1998; Xie and Ho 1999) also used time series approach to address software reliability prediction in the testing phase and operation phase.

3.2 General Theory of NHPP

Let $N(t)$ be the cumulative number of software failures by time t. The counting process $\{N(t),\ t \geq 0\}$ is said to be a NHPP with the intensity function $\lambda(t),\ t \geq 0$. The probability of exactly n failures occurring during the time interval $(0,\ t)$ for the NHPP is given by

$$P\ \{N(t) = n\} = \frac{[m(t)]^n}{n!} e^{-m(t)} \text{ for } n = 0,\ 1,\ 2, \ldots$$

where $m(t) = E[N(t)] = \int_0^t \lambda(s)ds$. $m(t)$ is the expected number of failures up to time t, which is also known as the mean value function (MVF).

Note that the forms of the MVF vary with different assumptions. In NHPP, the stationary assumption is relaxed compared with the Poisson process. In other words, $N(t)$ is Poisson-distributed with a time-dependent failure intensity function $\lambda(t)$, while the Poisson process holds the stationary assumption, $m(t) = \lambda t$.

A general NHPP model includes the following assumptions: (1) the failure process has an independent increment; (2) the failure rate of the process is given by $P\{N(t+\Delta t) - N(t) = 1\} = \lambda(t)\Delta t + o(\Delta t)$; (3) during a small interval Δt, the probability of more than one failure is negligible, that is $P\{N(t+\Delta t) - N(t) \geq 2\} = o(\Delta t)$, in which $o(\Delta t)$ represents a quantity that tends to be zero for a small Δt. The instantaneous failure intensity function $\lambda(t)$ is defined as

$$\lambda(t) = \lim_{\Delta t \to 0} \frac{R(t) - R(\Delta t + t)}{\Delta t R(t)} = \frac{f(t)}{R(t)}$$

where $R(t) = P[N(t) = 0] = e^{-m(t)}$.

The MVF is expressed as $m(t) = \int_0^t \lambda(s)ds$. One of the main objectives of NHPP software reliability model is to derive appropriate $m(t)$. The failure intensity function is equivalent to the derivative of MVF, which is $\lambda(t) = m'(t)$. Different assumption on the fault detection and fault removal process lead to different failure intensity function $\lambda(t)$. Reliability and other related measurements can be obtained by solving the differential equation $m'(t)$. The least-square estimate or MLE are commonly applied to estimate unknown parameters.

Software reliability $R(t)$ is defined as the probability that a software failure does not occur in $(0,\ t)$, that is

$$R(t) = P[N(t) = 0] = e^{-m(t)}$$

In general, during time interval $(t,\ t+x)$, software reliability can be described as

$$R(x|t) = P[N(t+x) - N(t) = 0] = e^{-[m(t+x) - m(t)]}$$

4 NHPP Software Reliability Models

NHPP has been successfully applied to model software reliability since Goel and Okumoto (1979b) firstly proposed their innovative model in 1979. Based on the development of Goel-Okumoto model, many NHPP software reliability models have been proposed in the past four decades to address different scenarios in software fault detection and fault correction processes, such as testing coverage, fault removal efficiency, fault dependency, time-delay fault removal, environmental factor on affecting software reliability, and multiple-release software. The failure processes are described by NHPP property with the MVF at time t, $m(t)$, and the failure intensity of the software program, $\lambda(t)$, which is also the derivative of MVF. Most existing NHPP software reliability models are developed based on the model below

$$\frac{dm(t)}{dt} = h(t)[N(t) - m(t)] \tag{1}$$

where $m(t)$ denotes the expected number of software failures by time t, $N(t)$ denotes the total number of fault content by time t, and $h(t)$ denotes the time-dependent fault detection rate per unit of time. The underlying assumption of Eq. (1) is the failure intensity is proportional to the residual fault content in the software. Depending on the model considerations, $N(t)$ and $h(t)$ can be modeled as a constant or a time-dependent function. Given many NHPP software reliability models are developed based on Eq. (1), we thus review these models depending on the focused scenarios in software development process. In the following sections, we rewrite the reviewed models based on the format of Eq. (1), and the coefficients of the reviewed models are nonnegative unless specified. The basic assumptions for the reviewed models are as follows unless specified: (1) software fault detection follows the NHPP; (2) the number of failures detected at any time t is proportional to the remaining faults in the software program; (3) when a failure is detected, the error that caused the failure is immediately removed; (4) all faults in a program are mutually independent from the perspective of failure detection.

4.1 NHPP Exponential Models

The Goel-Okumoto model (Goel and Okumoto 1979b) assumed that the isolated faults are removed prior to future test occasions and no new errors are introduced. The Goel-Okumoto model thus has $h(t) = b$ and $N(t) = a$ in Eq. (1) to obtain the MVF. Musa (1975) proposed a similar model to the Goel-Okumoto model by considering the relationship between execution time and calendar time, which assumed that $h(t) = c/nT$ and $N(t) = a$ in Eq. (1) to obtain the MVF, in which a is the number of failures in the program, c is the testing compression factor, T is the meaning time to failure at the beginning of the test, and n is the total number of possible failure during the maintained life of the program.

Ohba (1984) proposed the hyper-exponential growth model in consideration of different clusters of modules in a program. Each module contains a different initial number of errors and different failure rates, which are all assumed as constants in the software reliability model. It is well-known that the sum of an exponential distribution is a hyper-exponential distribution. Thus, the system software reliability model is more like the summation of each module's reliability model. Similar to the model proposed by Ohba (1984), Yamada and Osaki (1985) also proposed a software reliability model that considered software can be divided into K modules. The probability of faults for each module will be taken into consideration. The fault detection rate is the same within modules; however, it is various between modules. The total number of errors in the software is assumed as a constant and there are no new errors will be introduced during the fault detection process.

4.2 NHPP S-Shaped Models

In the NHPP S-shaped models, software reliability growth curve behaves as S-shaped. The curve crosses the exponential curve from below and the crossing occurs only once (Pham 2007). Time-dependent fault detection rate is applied in modeling software reliability growth trends. The concept of the S-shaped model is proposed to describe the changes of fault detection rate. Fault detection rate can be changed because of the difficulty level to detect different types of faults or the working experience that involves a learning process of software testers. Ohba et al. (1982) discussed a general NHPP model with S-shaped MVF in which $h(t)$ in Eq. (1) is treated as a time-dependent function. Ohba and Yamada (1984) proposed the NHPP model with the S-shaped MVF and considered the cumulative number of detected faults often seems to perform S-shaped. They stated that some of the faults are not detectable before some other faults are removed. The model proposed by Ohba and Yamada (1984) is called the inflection S-shaped model, which assumes $h(t) = b/(1+\beta e^{-bt})$ and $N(t) = a$ in Eq. (1), in which b and β represent failure detection rate and inflection factor, respectively.

Around the same time, Yamada et al. (1983, 1984, 1986) proposed several software reliability models considering the software fault detection process as a learning process. Specifically, when software testers get more familiar with the testing environment, specifications, and requirements, the fault detection rate will be higher. For example, Yamada et al. (1984) proposed a model, called the delayed S-shaped model, which assumed that $h(t) = b^2t/(bt + 1)$ and $N(t) = a$ in Eq. (1), in which b is the error detection rate per error in the steady-state. Nakagawa (1994) developed the connective NHPP model with S-curve forms. A group of modules called, main route modules, are tested first, followed by other modules. Even the failure intensity in the main route modules and other modules are similar, the failure growth curve performs as S-curve since the detection starts at different time points. Afterward, S-shaped reliability models are further developed in many studies (Chatterjee et al. 1997; Chatterjee and Singh 2014; Pham 1993; Pham and Zhang 1997; Pham et al. 1999). For example, Pham et al. (1999) developed the PNZ model in consideration of both the imperfect debugging and the learning effects, which assumed $h(t) = b/(1 + \beta e^{-bt})$ and $N(t) = a(1 + \alpha t)$ in Eq. (1), in which faults can be introducted during the debugging process at a constant rate of α, a is the total number of initial faults, and b and β have the same meanings as the model proposed by Ohba and Yamada (1984). Chatterjee and Singh (2014) incorporated a logistic-exponential testing coverage function in developing software reliability model, which assumed that $h(t) = c'(t)/(1 - c(t)) - d(t)$, $c(t) = \left(e^{bt} - 1\right)^k / \left[1 + \left(e^{bt} - 1\right)^k\right]$, $d(t) = \beta/(1 + \beta t)$, and $N(t) = a$ in Eq. (1), in which k is the positive shape parameter, b is a positive scale parameter, $d(t)$ is the fault introduction rate, modeled as a decreasing function of time.

4.3 NHPP Imperfect Debugging Models

NHPP perfect debugging models often assume when a failure occurs, the fault that caused the failure can be immediately removed, and no new faults are introduced (Goel and Okumoto 1979a; Ohba and Yamada 1984; Yamada et al. 1983; Hossain and Dahiya 1993), which means $N(t) = a$. Many models described in NHPP exponential models and S-shaped models are also NHPP perfect debugging models, in which $N(t)$ is modeled as a constant.

The concept of imperfect debugging is based on the assumptions (Pham 2000, 2007): (1) when the detected errors are removed, it is possible to introduce new errors; (2) the probability of finding an error in a program is proportional to the number of remaining errors in the program. Many reliability models are proposed based on NHPP imperfect debugging concept (Pham 2007, 1993; Yamada et al. 1984, 1991, 1992; Pham and Zhang 1997; Pham et al. 1999; Pham and Pham 2000; Inoue and Yamada 2004; Jones 1996; Kapur et al. 2007, 2011; Teng and Pham 2006; Tokuno and Yamada 2000; Fang and Yeh 2016; Xie and Yang 2003; Pham and Normann 1997). In the 1990s, Yamada et al. (1992) proposed two imperfect debugging models considering two types of fault content functions $N(t)$, which are $N(t) = \alpha e^{\beta t}$ and $N(t) = \alpha(1 + \gamma t)$, respectively, in which α is the number of initial fault content in the program prior to software testing, β and γ are the increasing rates of the number of the introduced faults to the program.

Software reliability models can belong to multiple categories, such as perfect debugging, imperfect debugging, S-shaped, exponential, testing effort, testing coverage, fault dependency, environmental factors, and software multiple-release. As an example, we review models that belong to both the categories of NHPP imperfect debugging and S-shaped models. S-shaped models were initially proposed to focus on the change of fault detection rate considering the difficulty level of detecting different types of software faults and the efficiency of detecting faults based on software testers' learning process, we therefore name the models that belong to both the categories of NHPP imperfect debugging and S-shaped models as NHPP imperfect debugging fault detection (IDFD) model. Besides the general assumptions of NHPP software reliability models, the generalized NHPP IDFD models also include the following assumptions: (1) the error detection rate differs among faults; (2) new faults are introduced during debugging.

Pham and Normann (1997) provided a generalized solution of Eq. (1). Specific MVF can be obtained by substituting different fault detection functions. The PNZ model (Pham et al. 1999) described in Sect. 4.2 also belongs to the category of the NHPP IDFD model. Moreover, Pham and Zhang (1997) proposed a model with $N(t) = c + a(1 - e^{-\alpha t})$ and $h(t) = b/(1 + \beta e^{-bt})$ in Eq. (1). Pham (2000, 2007) proposed a model by considering the fault introduction rate is an exponential function of the testing time, and the error detection rate follows a learning process, which assumed that $N(t) = \alpha e^{\beta t}$ and $b(t) = b/(1 + ce^{-bt})$ in Eq. (1). Later, Kapur et al. (2011) proposed two general frameworks for developing NHPP software reliability model in the presence of imperfect debugging and error generation. The first framework was formulated based on the assumption that there is no differentiation between failure observation and fault removal process. $h(t)$ and $N(t)$ in Eq. (1) are modeled as $h(t) = pF'(t)/(1 - F(t))$ and $N(t) = A + \alpha m(t)$, respectively, in which p is the probability of perfect debugging, $F(t)$ is the failure time distribution, A is the initial number of faults, and α is a constant fault introduction rate. The second framework is thus extended based on the assumption that there is a differentiation between failure observation and fault removal process.

4.4 NHPP Software Reliability Models on Software Testing

The common way to improve software reliability is to focus on in-house testing. Myers et al. (2011) defined software testing as a process of executing a program with the intent of finding errors. There are two fundamental rules in software testing. Firstly, it is intended to detect as many faults as possible during the in-house testing phase and remove the detected faults from the software system. Secondly, software failure data will be collected to predict system reliability, estimate the remaining faults, and schedule the product delivery date.

Owing to the fact that software debugging, testing, and verification are accounted for 50–70% of a software product's development cost. Indeed, software testing is always defined as a difficult and expensive section in software development (Ohmann and Liblit 2017; Hailpern and Santhanam 2002). Software debugging cost even goes higher if debugging is carried out in the operation phase. In practice, it is unlikely to release bug-free software products owning to its natural characteristics. Post-deployment failures are inevitable in complex software.

It is generally accepted that the longer time spent on software testing, the fewer faults that software will carry and the more reliable the software will be. However, this is not a practical approach. Exhaustive testing to execute all possible inputs unlikely to happen since too many possible combinations result in little improvement in system reliability (Weyuker 2004; Kaner et al. 2000). Moreover, full execution tracing is usually impractical for complex software programs due to the limitation of cost and resources (Ohmann and Liblit 2017). Furthermore, after software reaches a certain level of refinement, any further effort on removing faults will cause an exponentially increase in the total development cost but not much increase in reliability assessment (Pham and Zhang 1999a, b). Thus, how to test software efficiently and meet the predetermined reliability is a challenging task for both researchers and practitioners. In this section, we review NHPP software reliability models considering different scenarios in software testing, including testing coverage, testing efficiency, testing effort, time-delay fault removal, and multiple fault types.

Testing Coverage Models—Software systems have been widely applied in numerous safety-critical domains; however, large-scale software development is still considered a complicated and expensive activity. Since the testing phase plays an essential role in software development, a great number of software reliability models focus on the specific scenarios in the software development process, such as testing coverage, testing efficiency, testing resource allocation, and so on. Testing coverage is a measure that enables software developers to evaluate the quality of the tested software and determine how much additional effort is needed to improve the quality and reliability (Pham 2007). At the same time, the information on testing coverage can provide customers with a quantitative confidence criterion for software products. Pham and Zhang (2003) thus introduced a generalized model incorporating the measurement of testing coverage into software reliability assessment, in which $h(t) = c'(t)/(1 - c(t))$ in Eq. (1). This model indicates that the failure intensity depends on both the rate, the coverage rate *c'(t)*, and the percentage of the code that has not yet been covered by testing by time *t*, expressed as *1-c(t)*. Note that different functions of $N(t)$ and $c(t)$ can be plugged into Eq. (1) to obtain the MVF, given the formula of $h(t)$. One of the examples for the expressions of $c(t)$ and $N(t)$ is $c(t) = 1 - (1 + bt)e^{-bt}$ and $N(t) = a(1 + \alpha t)$. The model developed in Chatterjee and Singh (2014), reviewed in Sect. 4.2, is based on the model developed in Pham and Zhang (2003) by considering the fault introduction rate into $h(t)$, expressed as $h(t) = c'(t)/(1 - c(t)) - d(t)$, in which $d(t)$ is the fault introduction rate.

Inoue and Yamada (2004) proposed an alternative evaluation metric for the testing coverage in their study and further proposed a software reliability model by formulating the relationship between the alternative testing coverage evaluation function and the number of detected faults. The testing coverage measures are classified into several types, such as statement coverage, branch coverage, and path coverage. The measure of testing coverage is defined as the proportion of the number of statements that have been executed in the total number of statements. The software reliability proposed by Inoue and Yamada (2004) assumed that $h(t) = sc(t)$ and $N(t) = a$ in Eq. (1), in which $c(t) \equiv dC(t)/dt$, $C(t) = \alpha(1-e^{-b_{sta}t})/(1+ze^{-b_{sta}t})$, $z = (1-r)/r$, $r = b_{ini}/b_{sta}$, α is the target value of testing coverage to be attained, b_{ini} is the initial testing skill factor of the test case designers, and b_{sta} is the steady-state testing skill factor. Later, Li et al. (2008) incorporated logistic testing coverage function to develop a software reliability model. The time-varying test coverage function is expressed as $C(t) = C_{max}/(1 + Ae^{-at})$, in which C_{max} is the ultimate testing coverage that can be achieved by testing, a is the parameter of testing coverage increasing rate, and A is a constant. The proposed reliability model assumed that $N(t) = N$ and $h(t) = C'(t)/(1 - C(t))$ in Eq. (1).

Testing Efficiency Models—Section 4.3 reviewed software reliability models that addressed *new faults are introduced into debugging* based on the concept of imperfect debugging. Moreover, imperfect debugging can also be understood as the detected faults are removed at a certain rate instead of 100%. Jones (1996) stated that the faults removal efficiency (FRE) is an important factor in software quality and process management, which can provide software developers with the estimation of testing effectiveness and the prediction of additional effort. Note that FRE usually ranges from 15 to 50% for unit test, 25–40% for integration test, and 25–55% for system test (Pham 2007). Most software reliability models (Pham 2007) assumed that the detected faults are removed 100%. However, fault removal is not always 100% in practice. Some represented models are reviewed below. Zhang et al. (2003) proposed a generalized software reliability model based on imperfect debugging considering new faults can be introduced while debugging and the detected faults may not be removed completely. They defined the FRE in the study, which presented a new idea for later research. The FRE is defined as the percentage of bugs eliminated by reviews, inspections, and tests. Incorporating FRE into software reliability analysis will not only improve the prediction accuracy of software metrics but also define a tangible and quantifiable factor. The model proposed in Zhang et al. (2003) is expressed as $dm(t)/dt = b(t)[a(t) - pm(t)]$ and $da(t)/dt = \beta(t)dm(t)/dt$, in which p is the FRE, which means p percentage of detected faults can be completely eliminated during the debugging. Note that Zhang et al. (2003) provides a general solution for their proposed model and a specific solution with $b(t) = c/(1 + \alpha e^{-bt})$ and $\beta(t) = \beta$.

Kapur et al. (2007) proposed a software reliability model that incorporates testing efficiency regarding testing efforts in the testing phase and usage function in the testing phase. They (Kapur et al. 2007) assumed that: (1) when a software failure occurs, an instantaneous repair effort starts with the fault content is reduced by one with probability p and remains unchanged with probability $1 - p$; (2) the number of failures during the operation phase is dependent upon the usage function. Thus, their proposed model considers $h(t) = \left[pb/(1 + \beta e^{-bW(t)})\right]dW(t)/dt$ and $N(t) = a + \alpha m(t)$ in Eq. (1), in which $W(t)$ represents the cumulative testing effort in the time interval $(0, t]$. Base on the model proposed in Zhang et al. (2003), Li and Pham (2017) further proposed a software reliability model by incorporating FRE with $dm(t)/dt = h(t)[N(t) - pm(t)]$, $h(t) = \beta c'(t)/(1 - c(t))$, and $N(t) = a + \alpha m(t)$, in which β is proportionality constant and p is the FRE (same meaning as defined in Zhang et al. (2003)). Later, Zhu and Pham (2016) proposed a new way to formulate a software reliability model that addresses non-removed errors due to the experience of software testers, expressed as $dm(t)/dt = b(t)m(t)[1 - m(t)/L] - c(t)m(t)$, in which $b(t)$ is the fault detection rate per unit of time, L is the maximum number of faults existed in the program, and $c(t)$ is the non-removed error rate per unit of time. Zhu and Pham (2016) provided a general solution for the proposed model and a specific model with $b(t) = b/(1 + \beta e^{-bt})$, and $c(t) = c$.

Testing Effort Models—Yamada et al. (1991) proposed a software reliability model by using exponential and Rayleigh curves to describe the behavior of the amount of test effort spent on software testing. The proposed model (Yamada et al. 1991) is expressed as $dm(t)/dt = rw(t)[a - m(t)],\ 0 < r < 1$, and $w(t) = \alpha\beta e^{-\beta t}$ or $w(t) = \alpha\beta t e^{-\beta t^2/2}$, in which $w(t)$ is the test effort function representing the current test resource expenditures at testing time t, α and β are the coefficients associated with exponential and Rayleigh function. Huang and Kuo (2002) investigated a software reliability model based on the NHPP by incorporating a logistic testing effort function. They used the same base model (Yamada et al. 1991) and further proposed a new logistic testing effort function $w(t) = NA\alpha e^{-\alpha t}/\left[1 + Ae^{-\alpha t}\right]^2$, in which N is the total testing effort eventually consumed, α is the consumption rate of testing effort expenditures in the logistic testing effort function, and A is a constant parameter.

Huang and Lyu (2005) studied the impact of the testing effort and testing efficiency on modeling software reliability and the cost for optimal release time. The model proposed in Huang and Lyu (2005) used the same basic model (Yamada et al. 1991) and further considered a generalized logistic testing effort function $w(t) = N/\left[((k+1)/\beta)/(1 + Ae^{-\alpha kt})\right]^{1/k}$, in which N is the total amount of testing effort eventually consumed, k is the structuring index whose value is larger for better-structured software development efforts, A is the constant parameter in the logistic testing effort function, β is a normalized constant, and α is the consumption rate of testing effort expenditures in the logistic testing effort function. Huang (2005) further proposed a software reliability model incorporating the testing effort function in Huang and Lyu (2005) and the concept of change-point. Later, Lin and Huang (2008) incorporated the concept of multiple change-points into Weibull-type testing effort functions to propose a new software reliability model. Peng et al. (2014) proposed software reliability models in terms of fault detection process and fault correction process by incorporating testing effort function and imperfect debugging. Peng et al. (2014) assumed that $h(t) = b(t)w(t)$ for fault detection process, in which $b(t)$ is the fault detection rate per unit of testing effort at time t and $w(t)$ is the current testing effort expenditure at time t, and provided a general solution. Peng et al. (2014) also proposed the MVF for fault correction process with debugging delay $m(t) = \int_0^t \lambda_d(y)F(w(t) - W(y))dy$, in which $F(W(t) - W(y))$ is the probability that the fault detected at time y is corrected before time t, and $\lambda_d(t)$ is the fault intensity function of the fault detection process.

Time-delay Fault Removal Models—Time-delay fault removal models are also discussed in many studies. Xie and Zhao (1992) generalized Schneidewind's model by assuming a continuous time-dependent delay function which quantifies the expected delay in correcting the detected faults. Delay is treated as an increasing function of time t. The faults are easy to be corrected in the early stage of testing and become difficult to detect as time goes by. The MVF, $m_d(t)$, proposed in the fault detection process, is similar with Goel-Okumoto model, which is $m_d(t) = (\alpha/\beta)(1 - e^{-\beta t})$. The MVF, $m_c(t)$, proposed in the fault correct process is formulated as $m_c(t) = m_d(t - \Delta t),\ t \geq \Delta_t$.

Hwang and Pham (2009) developed a generalized NHPP software reliability model by considering quasi-renewal time-delay fault removal. They assumed that: (1) time-delay is defined as the interval between fault detection and fault removal; (2) time-delay is considered as a time-dependent function, described by a quasi-renewal process with parameter α and the first interarrival time s_1. This model provides a more relaxed assumption in software testing and debugging, which is very close to the practical testing and debugging process. Note that the testing resource allocation during the testing phase, which is usually depicted by the testing effort function, is affected not only by the fault detection rate but also the time to correct a detected fault. Moreover, Peng et al. (2014) not only incorporated the testing effort function and fault introduction into the fault detection process but also considered the debugging delay in the fault correction process.

Multiple Fault Types Models—Many studies stated that there exists more than one type of software fault in the program (Grottke et al. 2010; Laprie et al. 1990; Avizienis 1985; Grottke and Trivedi 2005, 2007; Shetti 2003; Alonso et al. 2013; Yazdanbakhsh et al. 2016; Deswarte et al. 1998; Laprie 1995). Different fault classes are categorized by practitioners and researchers to describe the characteristics of software faults that cause failures during the testing and operation phase (Grottke et al. 2010; Laprie et al. 1990; Yazdanbakhsh et al. 2016; Deswarte et al. 1998; Laprie 1995). The limits and challenges in the dependability of computer systems in terms of the fault class, such as physical faults, design faults, and interaction faults, are discussed in Yazdanbakhsh et al. (2016), Deswarte et al. (1998) as well. Ohba (1984) discussed two types of software faults, mutually independent faults, and mutually dependent faults. Tokuno and Yamada (2000) proposed an imperfect debugging software reliability model with two types of software failures involved. The first type is caused by the fault latent in the system, which is described by a geometrically decreasing function; the second type fault is randomly regenerated in the testing phase, which has a constant hazard rate. Lyu (1996) divided software failures into four groups according to the severity including catastrophic failure (a failure that may cause death or mission loss), critical failure (a failure that may cause severe injury or major system damage), marginal failure (a failure that may cause minor injury or degradation in mission performance), and minor failure (a failure that does not cause injury or system damage but may result in system failure and unscheduled maintenance).

Kapur and Younes (1995) considered the leading error and dependent error in model development. The expressions of $N(t)$ and $h(t)$ in Eq. (1) for the MVF, $m_1(t)$, for the leading error is written as $N(t) = q_1$ and $h(t) = b$. The expressions of $N(t)$ and $h(t)$ in Eq. (1) for the MVF, $m_2(t)$, for the dependent error is written as $N(t) = q_1$ and $h(t) = cm_1(t-T)/q$, in which c is the dependent error removal rate, T is the time-delay between the removal of the leading errors and the removal of the dependent errors, and $m_1(t-T)/q$ represents the ratio of the leading error removed to the initial error content at time t. Note that the cumulative number of software failures detected and removed by time t will be the summation of $m_1(t)$ and $m_2(t)$.

Pham (1996), Pham and Deng (2003) also studied multiple failure types with different detection rates. Three different types of errors are defined in Pham (1996), Pham and Deng (2003) including critical errors, which are very difficult to detect and remove; major errors, which are difficult to detect and remove; minor errors, which are easy to detect and remove. The $N(t)$ and $h(t)$ in Eq. (1) defined in Pham (1996) are $N(t) = n_i(t), h(t) = b_i$, and Eq. (1) is reformulated as $dn_i(t)/dt = \beta_i dm_i(t)/dt$, in which i represents different types of errors defined and β_i represents the type i error introduction rate that satisfies $0 \leq \beta_i \leq 1$. Later, Pham and Deng (2003) further considered the expression of $h(t)$ can be modeled as a non-decreasing S-shaped model, in which $h(t) = b_i/(1 + \theta_i e^{-b_i t})$. Huang and Lin (2006) incorporated fault dependence and delay debugging in the software reliability growth model. Huang and Lin (2006) considered all detected faults can be categorized as either leading faults or dependent faults. For the leading faults, the expressions of $N(t)$ and $h(t)$ in Eq. (1) are $N(t) = a_1$ and $h(t) = r$, in which a_1 is the total number of leading faults. For the dependent faults, the expressions of $N(t)$ and $h(t)$ in Eq. (1) are $N(t) = a_2$ and $h(t) = \theta m_1(t - \varphi(t))/a$, in which $m_1(t)$ is the expected number of leading errors, θ is the fault detection rate of dependent faults, a is the total number of initial faults, and $\varphi(t)$ is the delay-effect factor.

Grottke et al. (2010) studied the proportion of the various fault types including Bohrbugs, non-aging-related Mandel bugs, aging-related bugs, and unknown bugs and their evolvement with time based on the fault discovered in the onboard software for 18 JPL/NASA space missions. However, they did not provide a quantitative way to estimate the number of faults. Zhu and Pham (2017a) proposed a new NHPP software reliability model by considering software fault dependency and imperfect fault removal. Two types of software faults are defined, Type I (independent) fault and Type II (dependent) fault, based on the consideration of fault dependency. The assumptions in Zhu and Pham (2017a) are: (1) Type I fault is detected and removed in Phase I. Type II fault is detected and removed in Phase II. The un-removed Type I faults from Phase I are still not able to detect in Phase II; (2) in both phases, there exists a certain portion of software faults that the software development team is not able to remove. In Phase I, the MVF of Type I fault is expressed as $dm_1(t)/dt = b_1(t)[a_1(t) - m_1(t)] - c_1(t)m_1(t),\ t \leq t_0$, in which $a_1(t)$ is total software content, $b_1(t)$ is Type I fault detection rate, $c_1(t)$ is the non-removable fault rate in Phase I. In Phase II, the MVF of Type II fault is expressed as $dm_2(t)/dt = (b_2(t)/a_2(t))m_2(t)[a_2(t) - m_2(t)] - c_2(t)m_2(t),\ t > t_0$, in which $a_2(t)$ is total software content in Phase II, $b_2(t)$ is Type II fault detection rate, and $c_2(t)$ is the non-removable fault rate in Phase II.

4.5 NHPP Multiple-Release Software Reliability Models

As software development moves further away from the rigid and monolithic model, the importance of software multiple-release is brought to the vanguard. It is unlikely to deliver all features that customers wanted in the single release because of the limited budget, unavailable resources, estimated risk, and constrained working schedules. Staying competitive in the market and keeping profitable for a software product is difficult with having only a single release especially when rival releases a new release carrying more attractive features and satisfying more customer requirements (Saliu and Ruhe 2005). As a result of multiple releases planning, software organization will have more competitive and overwhelming advantages to balance the competing stakeholder's demands and benefits according to the available resource (Ruhe and Momoh 2005; Svahnberg et al. 2010). On the other hand, a large software system continually desires to align with the changing customer requirements for the sake of market share. In order to obtain feedback from users, figure out what customers really look for, and assign a lower software development cost, a certain portion of increments on the requirements for a multiple-release product is essential for the growth of an organization (Maurice et al. 2006; Greer and Ruhe 2004; Missbauer 2002). Thus, software organization needs to modify parts of the existing modules to extend the current functionality, usability, and understandability by adding new features and correcting the problems from the previous releases (Al-Emran and Pfahl 2007; Gorschek and Davis 2008). Additionally, agile software development is getting more attention in recent years. Agile is an iterative and team-based approach, which emphasizes the rapid delivery of an application in complete functional components (Lotz 2018). The wide adoption of agile methodology also promotes software multiple-release.

Furthermore, most software products are not introduced into the market with full capacities at their initial release. New features will be added and existing features will be enhanced after launched software for a while. Hence, software multiple-release is critical to keeping a software product stays competitive in the market. Modeling and predicting software failure behaviors for single-release software systems have been extensively studied in the past few decades. However, only a few researchers studied multiple-release software reliability and introduced prediction models to explain software fault detection process and fault removal process for multiple-release software.

Garmabaki et al. (2011) incorporated different severities level used to describe the difficulty of correcting faults in the upgrade process to develop a multi up-gradation software reliability model. Faults are classified into two categories, simple fault, and hard fault. The fault removal for the development of the new release depends on the fault from the previous releases and the fault generated in that release. Hu et al. (2011) considered the effect of multiple releases regarding the fault detection process in software development. They assumed that there is no gap between the release of the previous version and the development of the next version. In this work, in order to study the effects of multiple releases on the fault dynamics during the whole software development, they considered a scenario where a software development team develops, tests, and releases software version by version. The field test of each version continues after its release so that faults can continue to be detected and corrected until the next version is ready to be tested. In case a fault is detected in the field test of any version, it will be reported and corrected in both the current version and its subsequent version.

Kapur et al. (2012) introduced the combined effect of schedule pressure and resource limitations by the use of the Cobb–Douglas production function in software reliability modeling. The Cobb–Douglas function illustrates the total production output can be obtained by the amount of labor input, capital input, and total factor productivity. An optimal release planning problem is formulated in this study for software with multiple releases with the solution obtained by applying the genetic algorithm method. Yang et al. (2016) incorporated fault detection and fault correction process in multiple-release software reliability modeling. They considered there is a time-delay in fault repair after detecting faults. The time-delay function is explained by an exponential function or a gamma function. They also assumed the faults in a new version including both the undetected faults from the last version and the newly introduced faults during the development process of the new version. Pachauri et al. (2015) proposed a software reliability growth model by considering fault reduction factor (FRF) and extended this idea to multiple-release software systems. FRF is defined as the ratio of the total number of reduced faults to the total number of failures. FRF is not a constant, which can be affected by other factors, such as resources allocation.

The multiple-release software reliability models reviewed above mainly focused on obtaining optimal release time by optimizing the software cost model without considering the dependent relationship of software faults generated from different releases. Therefore, Zhu and Pham (2017b) focused on the development of a multiple-release software reliability model considering the remaining software faults from the previous releases and the newly introduced faults resulting from the newly introduced features in the development of the next release. Additionally, the dependent fault detection process is taken into account in this research. In particular, the detection of a new software fault for developing the next release depends on the detection of the remaining faults from the previous releases and the detection of the newly introduced faults. They further discussed the behaviors of the proposed software reliability model through mathematical proofs.

4.6 NHPP Environmental Factor Based Software Reliability Models

Software development process has gone through a great change during the past one and half decades. The rise of the Internet had led to rapid growth in the demand for international information display and email systems on the World Wide Web. Software programmers are required to handle various illustrations, maps, photographs, and other images, plus simple animations at a rate we have never seen before. The high technology has an ever-increasing impact on daily life, which drives the software release cycle to become shorter than before, for instance, many companies have shortened their software release cycle from traditional 18 months to 3 months, in order to respond to the fast-changing and competitive market (HP Applications Handbook 2012; Khomh et al. 2012). Moreover, as high technology gets more involved in our everyday life, there are a wide variety of computational devices like mobile phones, tablet PCs, laptops, desktops, and notebooks (Gallud et al. 2012), which also brings more challenges to software developers, such as application maintenance, device consistency, and dynamic version settings (Eisenstein et al. 2001). Customers also have more requirements on the specific design and functionality of the software product. A user-friendly interface, involved in the interaction amongst users, designers, hardware systems, and software systems, has been emphasized to a great extent nowadays. Furthermore, for practitioners and researchers, programming skills, programming language, domain knowledge, and even the programmer organization and team size are different compared with a decade ago. Finally, software development is distributed across multiple locations as the development of globalization (Ramasubbu and Balan 2007). However, such cross-site work patterns may take a much longer time and require much more effort, even though the work size and complexity are similar (Herbsleb et al. 2000, 2001; Herbsleb and Mockus 2003).

Given the current trends of the software development process, which are the adoption of software product lines, software development globalization, and the establishment of software ecosystems, the complicated and human-centered software development process needs to be addressed more appropriately. Meanwhile, environmental factors play significant impacts on affecting software reliability during the software development process (Zhang and Pham 2000; Zhang et al. 2001; Zhu et al. 2015; Zhu and Pham 2017c; Misra et al. 2009; Chow and Cao 2008; Clarke and O'Connor 2012; Sawyer and Guinan 1998; Roberts et al. 1998). Indeed, how to define and incorporate single/multiple environmental factors that present a significant impact on reliability into the software reliability model is critical to address modern software development in practice.

Environmental Factors in Software Development—Although no general definition has been given to defining what are the environmental factors affecting software reliability during the software development process, there have been many related works that defined different types of factors in software development from various perspectives.

Zhang and Pham (2000) defined 32 environmental factors and characterized the impacts of these environmental factors affecting software reliability during the software development process for single-release software. These 32 environmental factors are defined from the four phases of software development, general information, and the interaction with hardware systems. Software development is divided into four phases in this study: analysis phase, design phase, coding phase, and testing phase. The authors conducted a survey investigation and obtained empirically quantitative and qualitative data from managers, software engineers, designers, programmers, and testers, who participated in software development practices. This study also identified the important environmental factors in software development and analyzed the correlations between these environmental factors. Later, Zhang et al. (2001) provided an exploratory analysis to further analyze the detailed relationships of these environmental factors. Zhu et al. (2015) revisited these 32 environmental factors defined in Zhang and Pham (2000) and analyzed their impacts on software reliability during software development based on a current survey distributed to software development practitioners. As the application of agile development and the increasing popularity of multiple-release software products in many organizations, Zhu and Pham (2017c) further conducted another study to investigate the impact level of these 32 environmental factors on affecting software reliability in the development of multiple-release software to provide a sound and concise guidance to software practitioners and researchers.

Sawyer and Guinan (1998) presented the effects on software development performance that depend on the production method of software development and the social process of how people work together in the software development environment. Roberts et al. (1998) proposed five factors that are essential to implement a system development methodology, including organizational system development methodology transition, functional management involvement/support, use of models, and external support. Chow and Cao (2008) collected the survey data from 109 agile projects from a diverse group of organizations with different sizes, industries, and geographic locations to provide empirical information for the statistical analysis. Based on the multiple regression analysis, the critical success factors are identified as a correct delivery strategy, a proper practice of agile software engineering techniques, and a high-caliber team. Three other factors that could be critical to certain success dimensions are identified as a good agile project management process, an agile-friendly team environment, and strong customer involvement.

Misra et al. (2009) conducted a large-scale survey-based study to identify the success factors from the perspective of agile software development practitioners who have successfully adopted agile software development in their projects. This study identified nine out of the fourteen hypothesized factors that have statistically significant relationships with "success". The important success factors are customer satisfaction, customer collaboration, customer commitment, decision time, corporate culture, control, personal characteristics, societal culture, and training and learning. Clarke and O'Connor (2012) researched the situational factors affecting the software development process. Rigorous data coding techniques from Grounded Theory have been applied in this study. They concluded that the resulting reference framework of situational factors consists of eight classifications and 44 factors that inform the software process. On the other hand, this framework also provides useful information for practitioners who are challenged with defining and maintaining the software development process.

Environmental Factor based Software Reliability Models—Only a few studies incorporated environmental factors, the random effect of the testing/operating environments, or other factors, such as FRF that could be influenced by many environmental factors, to develop software reliability models.

Teng and Pham (2006) presented a new methodology for predicting software reliability in the field environment. A generalized random field environment (RFE) software reliability model which can cover both the testing phase and operating phase is proposed in this study by assuming all the random effect in the field environments can be captured by a unit-free environmental factor. Two specific RFE software reliability models are developed by the use of the generalized RFE software reliability model, called the γ-RFE model and the β-RFE model, to describe different random effects in the operation phase. Hsu et al. (2011) integrated the FRF into software reliability models. The FRF is proposed by Musa (1975), which is generally defined as the ratio of net fault reduction to failure experience (Musa 1980; Musa et al. 1987), which could be influenced by many environmental factors, such as fault dependency, human learning process, imperfect debugging, and delay debugging. The authors firstly studied the trend of the FRF and considered it as a time-variable function, and then incorporated the FRF in software reliability growth modeling to improve the accuracy of failure prediction. Pachauri et al. (2015) also considered the impact of FRF in developing a software reliability growth model. Pham (2014) incorporated the uncertainty of the operating environments into a software Vtub-shaped fault detection rate model. In particular, the fault detection rate in this study is represented by a Vtub-shape function, and the uncertainty of the operating environments is represented by a random variable, modeled as a gamma distribution.

With the recent investigations of the significance of environmental factors in software development, Zhu and Pham (2018) incorporated one of the top 10 significant environmental factors from Zhu et al. (2015), Zhu and Pham (2017c), Percentage of Reused Modules (PoRM), to be a random variable which has a random effect on fault detection rate. This study introduced the Martingale framework, specifically, Brownian motion and white noise process into the stochastic fault detection process, which is used to model the impact resulting from the randomness of PoRM. They further proposed a single-environmental-factor software reliability model considering the gamma-distributed PoRM and the randomness associated with PoRM. Later, considering the significance of the impacts from multiple environmental factors (Zhu et al. 2015; Zhu and Pham 2017c), Zhu and Pham (2020) proposed a generalized software reliability model with multiple environmental factors and the associated randomness under the Martingale framework. The randomness is reflected in the process of detecting software faults. Indeed, this is a stochastic fault detection process. Software practitioners and researchers are able to obtain a specific multiple-environmental-factors software reliability model according to the individual application environments from the proposed generalized multiple-environmental-factors software reliability model.

5 Conclusion

Given our modern societies are increasingly dependent on software systems, such as transportation networks, smart grids, and healthcare systems, software systems malfunction can result in cascading failures. Meanwhile, large-scale software development is still a complex, effort-consuming, and expensive activity. The consequences of software failures thus become costly and even dangerous. In this chapter, we review probabilistic software reliability models with different groups. Considering the wide adoption of NHPP based software reliability models in practical software reliability engineering, this chapter mainly focuses on the review of NHPP based software reliability models that address various concerns in software development practices, such as testing efficiency, testing coverage, multiple fault types, time-delay fault removal, and environmental factors.

References

Al-Emran A, Pfahl D (2007) Operational planning, re-planning and risk analysis for software releases. In: International conference on product focused software process improvement. Springer, pp 315–329

Alonso J, Grottke M, Nikora AP, Trivedi KS (2013) An empirical investigation of fault repairs and mitigations in space mission system software. In: Proceedings of 2013 43rd annual IEEE/IFIP international conference on dependable systems and networks (DSN). IEEE, pp 1–8

Avizienis A (1985) The N-version approach to fault-tolerant software. IEEE Trans Softw Eng 12:1491–1501

Bastani FB, Ramamoorthy CV (1986) Input-domain-based models for estimating the correctness of process control programs. Reliab Theory 321–378

Belady LA, Lehman MM (1976) A model of large program development. IBM Syst J 15(3):225–252

Bosch J, Bosch-Sijtsema P (2010) From integration to composition: on the impact of software product lines, global development and ecosystems. J Syst Softw 83(1):67–76

Cai KY (1998) On estimating the number of defects remaining in software. J Syst Softw 40(2):93–114

Cascio WF, Shurygailo S (2003) E-leadership and virtual teams. Organ Dyn 31(4):362–376

Catelani M, Ciani L, Scarano VL, Bacioccola A (2011) Software automated testing: a solution to maximize the test plan coverage and to increase software reliability and quality in use. Comput Stand Interfaces 33(2):152–158

Chang IH, Pham H, Lee SW, Song KY (2014) A testing-coverage software reliability model with the uncertainty of operating environments. International Journal of Systems Science: Operations & Logistics 1(4):220–227

Chatterjee S, Singh JB (2014) A NHPP based software reliability model and optimal release policy with logistic–exponential test coverage under imperfect debugging. Int J Syst Assur Eng Manag 5(3):399–406

Chatterjee S, Misra RB, Alam SS (1997) Prediction of software reliability using an auto regressive process. Int J Syst Sci 28(2):211–216

Chow T, Cao DB (2008) A survey study of critical success factors in agile software projects. J Syst Softw 81(6):961–971

Clarke P, O'Connor RV (2012) The situational factors that affect the software development process: towards a comprehensive reference framework. Inf Softw Technol 54(5):433–447

Clements P, Northrop L (2002) Software product lines. Addison-Wesley

COMPUTERWORLD (2020) Top software failures in recent history. https://www.computerworlduk.com/galleries/infrastructure/top-software-failures-recent-history-3599618/#22

Coutinho JDS (1973) Software reliability growth. In: Proceedings of the IEEE symposium on computer software reliability, pp 58–64

Dai YS, Xie M, Poh KL (2005) Modeling and analysis of correlated software failures of multiple types. IEEE Trans Reliab 54(1):100–106

Dang Y, Ge S, Huang R, Zhang D (2011) Code clone detection experience at Microsoft. In: Proceedings of the 5th international workshop on software clones. ACM, pp 63–64

Deswarte Y, Kanoun K, Laprie JC (1998) Diversity against accidental and deliberate faults. In: Proceedings of the computer security, dependability and assurance: from needs to solution. IEEE, pp 171–181

Eisenstein J, Vanderdonckt J, Puerta A (2001) Applying model-based techniques to the development of UIs for mobile computers. In: Proceedings of the 6th international conference on intelligent user interfaces. ACM, pp 69–76

Fang CC, Yeh CW (2016) Effective confidence interval estimation of fault-detection process of software reliability growth models. Int J Syst Sci 47(12):2878–2892

Febrero F, Calero C, Moraga MÁ (2016) Software reliability modeling based on ISO/IEC SQuaRE. Inf Softw Technol 70:18–29

Gallud JA, Peñalver A, López-Espín JJ, Lazcorreta E, Botella F, Fardoun HM, Sebastián G (2012) A proposal to validate the user's goal in distributed user interfaces. Int J Hum Comput Interact 28(11):700–708

Garmabaki AH, Aggarwal AG, Kapur PK (2011) Multi up-gradation software reliability growth model with faults of different severity. In: Proceedings of 2011 IEEE international conference on industrial engineering and engineering management (IEEM). IEEE, pp 1539–1543

Goel AL (1985) Software reliability models: assumptions, limitations, and applicability. IEEE Trans Software Eng 12:1411–1423

Goel AL, Okumoto K (1979a) A Markovian model for reliability and other performance measures of software systems. In: National computer conference. IEEE, pp 769–774

Goel AL, Okumoto K (1979b) Time-dependent error-detection rate model for software reliability and other performance measures. IEEE Trans Reliab 28(3):206–211

Gorschek T, Davis AM (2008) Requirements engineering: in search of the dependent variables. Inf Softw Technol 50(1–2):67–75

Goseva-Popstojanova K, Trivedi K (1999) Failure correlation in software reliability models. In: Proceeding of the 10th international symposium on software reliability engineering (ISSRE). IEEE, pp 232–241

Greer D, Ruhe G (2004) Software release planning: an evolutionary and iterative approach. Inf Softw Technol 46(4):243–253

Grottke M, Trivedi KS (2005) A classification of software faults. J Reliab Eng Assoc Jpn 27(7):425–438

Grottke M, Trivedi KS (2007) Fighting bugs: remove, retry, replicate, and rejuvenate. Computer 40(2)

Grottke M, Nikora AP, Trivedi KS (2010) An empirical investigation of fault types in space mission system software. In: Proceedings of 2010 IEEE/IFIP international conference on dependable systems and networks (DSN). IEEE, pp 447–456

Hailpern B, Santhanam P (2002) Software debugging, testing, and verification. IBM Syst J 41(1):4–12

Hallsteinsen S, Hinchey M, Park S, Schmid K (2008) Dynamic software product lines. Computer 41(4)

Halstead MH (1977) Elements of software science. Elsevier

Han S, Dang Y, Ge, S., Zhang D, Xie T (2012) Performance debugging in the large via mining millions of stack traces. In: Proceedings of the 34th international conference on software engineering. IEEE, pp 145–155

Hartz MA, Walker EL, Mahar D (1997) Introduction to software reliability: a state of the art review. The Center

Herbsleb JD, Mockus A (2003) An empirical study of speed and communication in globally distributed software development. IEEE Trans Softw Eng 29(6):481–494

Herbsleb JD, Mockus A, Finholt TA, Grinter RE (2000) Distance, dependencies, and delay in a global collaboration. In: Proceedings of the 2000 ACM conference on computer supported cooperative work. ACM, pp 319–328

Herbsleb JD, Mockus A, Finholt TA, Grinter RE (2001) An empirical study of global software development: distance and speed. In: Proceedings of the 23rd international conference on software engineering. IEEE Computer Society, pp 81–90

Ho SL, Xie M (1998) The use of ARIMA models for reliability forecasting and analysis. Comput Ind Eng 35(1–2):213–216

Hossain SA, Dahiya RC (1993) Estimating the parameters of a non-homogeneous Poisson-process model for software reliability. IEEE Trans Reliab 42(4):604–612

HP Applications Handbook (2012) Shorten release cycles by bringing developers to application lifecycle management. http://www.hp.com/hpinfo/newsroom/press_kits/2011/optimization2011/Lifecycle.pdf

Hsu CJ, Huang CY, Chang JR (2011) Enhancing software reliability modeling and prediction through the introduction of time-variable fault reduction factor. Appl Math Model 35(1):506–521

Hu QP, Peng R, Xie M, Ng SH, Levitin G (2011) Software reliability modelling and optimization for multi-release software development processes. In: Proceedings of2011 IEEE international conference on industrial engineering and engineering management (IEEM). IEEE, pp 1534–1538

Huang X (1984) The hypergeometric distribution model for prediction the reliability of software. Microelectron Reliab 24(1)

Huang CY (2005) Performance analysis of software reliability growth models with testing-effort and change-point. J Syst Softw 76(2):181–194

Huang CY, Kuo SY (2002) Analysis of incorporating logistic testing-effort function into software reliability modeling. IEEE Trans Reliab 51(3):261–270

Huang CY, Lin CT (2006) Software reliability analysis by considering fault dependency and debugging time lag. IEEE Trans Reliab 55(3):436–450

Huang CY, Lyu MR (2005) Optimal release time for software systems considering cost, testing-effort, and test efficiency. IEEE Trans Reliab 54(4):583–591

Hwang S, Pham H (2009) Quasi-renewal time-delay fault-removal consideration in software reliability modeling. IEEE Trans Syst Man Cybern Part A Syst Hum 39(1):200–209

Inoue S, Yamada S (2004) Testing-coverage dependent software reliability growth modeling. Int J Reliab Qual Saf Eng 11(04):303–312

Jansen S, Brinkkemper S, Finkelstein A (2007) Providing transparency in the business of software: a modeling technique for software supply networks. Establishing the foundation of collaborative networks. Springer, pp 677–686

Jelinski Z, Moranda P (1972) Software reliability research. Stat Comput Perform Eval 465–484

Jones C (1996) Software defect-removal efficiency. Computer 29(4):94–95

Kaner C, Falk J, Nguyen HQ (2000) Testing computer software, 2nd edn. Dreamtech Press

Kapur PK, Younes S (1995) Software reliability growth model with error dependency. Microelectron Reliab 35(2):273–278

Kapur PK, Gupta A, Jha PC (2007) Reliability analysis of project and product type software in operational phase incorporating the effect of fault removal efficiency. Int J Reliab, Qual Saf Eng 14(03):219–240

Kapur PK, Pham H, Anand S, Yadav K (2011) A unified approach for developing software reliability growth models in the presence of imperfect debugging and error generation. IEEE Trans Reliab 60(1):331–340

Kapur PK, Pham H, Aggarwal AG, Kaur G (2012) Two dimensional multi-release software reliability modeling and optimal release planning. IEEE Trans Reliab 61(3):758–768

Khomh F, Dhaliwal T, Zou Y, Adams B (2012) Do faster releases improve software quality? An empirical case study of Mozilla Firefox. In: Proceedings of the 9th IEEE working conference on mining software repositories. IEEE, pp 179–188

Laprie JC (1995) Dependable computing: concepts, limits, challenges. In: Special issue of the 25th international symposium on fault-tolerant computing, pp 42–54

Laprie JC, Arlat J, Beounes C, Kanoun K (1990) Definition and analysis of hardware-and software-fault-tolerant architectures. Computer 23(7):39–51

Lehman MM (1980) Programs, life cycles, and laws of software evolution. Proc IEEE 68(9):1060–1076

Li Q, Pham H (2017) A testing-coverage software reliability model considering fault removal efficiency and error generation. PloS One 12(7):e0181524

Li H, Li Q, Lu M (2008) Software reliability modeling with logistic test coverage function. In: Proceedings of the 19th international symposium on software reliability engineering (ISSRE). IEEE, pp 319–320

Lin CT, Huang CY (2008) Enhancing and measuring the predictive capabilities of testing-effort dependent software reliability models. J Syst Softw 81(6):1025–1038

Littlewood B (1979) Software reliability model for modular program structure. IEEE Trans Reliab 28(3):241–246

Lotz M (2018) Waterfall vs. Agile: which is the right development methodology for your project? https://www.seguetech.com/waterfall-vs-agile-methodology/

Lyu M (1996) Handbook of software reliability engineering. IEEE Computer Society Press, McGraw-Hill

Lyu MR (2007) Software reliability engineering: a roadmap. In: 2007 future of software engineering. IEEE Computer Society, pp 153–170

Maurice S, Ruhe G, Saliu O (2006) Decision support for value-based software release planning. Value-based software engineering. Springer, pp 247–261

McCabe TJ (1976) A complexity measure. IEEE Trans Softw Eng 4:308–320

Mellor P (1987) Software reliability modelling: the state of the art. Inf Softw Technol 29(2):81–98

Messerschmitt DG, Szyperski C (2005) Software ecosystem: understanding an indispensable technology and industry. MIT Press

Miller DR, Sofer A (1985) Completely monotone regression estimates of software failure rates. In: Proceedings of the 8th international conference on software engineering. IEEE Computer Society, pp 343–348

Mills H (1972) On the statistical validation of compute programs. IBM federal systems division report, 72-6015

Misra SC, Kumar V, Kumar U (2009) Identifying some important success factors in adopting agile software development practices. J Syst Softw 82(11):1869–1890

Missbauer H (2002) Aggregate order release planning for time-varying demand. Int J Prod Res 40(3):699–718

Moranda PB (1981) An error detection model for application during software development. IEEE Trans Reliab 30(4):309–312

Moravec H (1998) When will computer hardware match the human brain. J Evol Technol 1(1):10

Musa JD (1975) A theory of software reliability and its application. IEEE Trans Softw Eng 1(03):312–327

Musa JD (1980) The measurement and management of software reliability. Proceedings of the IEEE 68(9):1131–1143

Musa JD, Iannino A, Okumoto K (1987) Software reliability: measurement, prediction, application, McGraw-Hill

Musa JD, Iannino A, Okumoto K (1990) Software reliability. Adv Comput 30:85–170

Myers GJ, Sandler C, Badgett T (2011) The art of software testing. Wiley

Nakagawa Y (1994) A connective exponential software reliability growth model based on analysis of software reliability growth curves. IEICE Trans 77:433–442

Ohba M (1984) Software reliability analysis models. IBM J Res Dev 28(4):428–443

Ohba M, Yamada S (1984) S-shaped software reliability growth models. In: Proceedings of the 4th international colloquium on reliability and maintainability, pp 430–436

Ohba M, Yamada S, Takeda K, Osaki S (1982) S-shaped software reliability growth curve: how good is it? In: Proceedings of COMPSAC'82, pp 38–44

Ohmann P, Liblit B (2017) Lightweight control-flow instrumentation and postmortem analysis in support of debugging. Autom Softw Eng 24(4):865–904

Pachauri B, Dhar J, Kumar A (2015) Incorporating inflection S-shaped fault reduction factor to enhance software reliability growth. Appl Math Model 39(5–6):1463–1469

Patterson DA, Hennessy JL (2013) Computer organization and design: the hardware/software interface. Morgan Kaufmann Publishers

Peng R, Li YF, Zhang WJ, Hu QP (2014) Testing effort dependent software reliability model for imperfect debugging process considering both detection and correction. Reliab Eng Syst Saf 126:37–43

Pham H (1993) Software reliability assessment: imperfect debugging and multiple failure types in software development. EGandG-RAAM-10737. Idaho National Engineering Laboratory

Pham H (1996) A software cost model with imperfect debugging, random life cycle and penalty cost. Int J Syst Sci 27(5):455–463

Pham H (2000) Software reliability. Springer Science & Business Media

Pham, H. (2007). System Software Reliability. Springer Science & Business Media.

Pham H (2014) A new software reliability model with Vtub-shaped fault-detection rate and the uncertainty of operating environments. Optimization 63(10):1481–1490

Pham H, Deng C (2003) Predictive-ratio risk criterion for selecting software reliability models. In: Proceedings of the 9th international conference on reliability and quality in design, pp 17–21

Pham H, Normann L (1997) A generalized NHPP software reliability model. In: Proceeding of 3rd ISSAT international conference on reliability and quality in design, Aug 1997
Pham L, Pham H (2000) Software reliability models with time-dependent hazard function based on Bayesian approach. IEEE Trans Syst Man Cybern Part A Syst Hum 30(1):25–35
Pham H, Zhang X (1997) An NHPP software reliability model and its comparison. Int J Reliab Qual Saf Eng 4(03):269–282
Pham H, Zhang X (1999a) Software release policies with gain in reliability justifying the costs. Ann Softw Eng 8(1–4):147–166
Pham H, Zhang X (1999b) A software cost model with warranty and risk costs. IEEE Trans Comput 48(1):71–75
Pham H, Zhang X (2003) NHPP software reliability and cost models with testing coverage. Eur J Oper Res 145(2):443–454
Pham H, Nordmann L, Zhang Z (1999) A general imperfect-software-debugging model with S-shaped fault-detection rate. IEEE Trans Reliab 48(2):169–175
Ramasubbu N, Balan RK (2007) Globally distributed software development project performance: an empirical analysis. In: Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering. ACM, pp 125–134
Roberts TL, Gibson ML, Fields KT, Rainer RK (1998) Factors that impact implementing a system development methodology. IEEE Trans Softw Eng 24(8):640–649
Ruhe G, Momoh J (2005) Strategic release planning and evaluation of operational feasibility. In: Proceedings of the 38th Annual Hawaii international conference on system science. IEEE, pp 313b-313b.
Saliu O, Ruhe G (2005) Software release planning for evolving systems. Innov Syst Softw Eng 1(2):189–204
Sawyer S, Guinan PJ (1998) Software development: processes and performance. IBM Syst J 37(4):552–569
Schick GJ, Wolverton RW (1978) An analysis of competing software reliability models. IEEE Trans Softw Eng 2:104–120
Shetti NM (2003) Heisenbugs and Bohrbugs: why are they different. Techn. Ber. Rutgers, The State University of New Jersey
Singpurwalla ND, Soyer R (1985) Assessing (software) reliability growth using a random coefficient autoregressive process and its ramifications. IEEE Trans Softw Eng 12:1456–1464
Svahnberg M, Gorschek T, Feldt R, Torkar R, Saleem SB, Shafique MU (2010) A systematic review on strategic release planning models. Inf Softw Technol 52(3):237–248
Tassey G (2002) The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology, RTI Project, 7007(011)
Teng X, Pham H (2006) A new methodology for predicting software reliability in the random field environments. IEEE Trans Reliab 55(3):458–468
Tohma Y, Yamano H, Ohba M, Jacoby R (1991) The estimation of parameters of the hypergeometric distribution and its application to the software reliability growth model. IEEE Trans Softw Eng 17(5):483–489
Tokuno K, Yamada S (2000) An imperfect debugging model with two types of hazard rates for software reliability measurement and assessment. Math Comput Model 31(10–12):343–352
Wall JK, Ferguson PA (1977) Pragmatic software reliability prediction. In: Annual reliability and maintainability symposium, pp 485–488
Weyuker EJ (2004) How to judge testing progress. Inf Softw Technol 46(5):323–328
Xie M (1991) Software reliability modelling. World Scientific
Xie M, Ho SL (1999) Analysis of repairable system failure data using time series models. J Qual Maint Eng 5(1):50–61
Xie M, Yang B (2003) A study of the effect of imperfect debugging on software development cost. IEEE Trans Softw Eng 29(5):471–473

Xie M, Zhao M (1992) The Schneidewind software reliability model revisited. In: Proceedings of the 3rd international symposium on software reliability engineering (ISSRE). IEEE, pp 184–192

Yamada S, Osaki S (1985) Software reliability growth modeling: models and applications. IEEE Trans Softw Eng 12:1431–1437

Yamada S, Ohba M, Osaki S (1983) S-shaped reliability growth modeling for software error detection. IEEE Trans Reliab 32(5):475–484

Yamada S, Ohba M, Osaki S (1984) S-shaped software reliability growth models and their applications. IEEE Trans Reliab 33(4):289–292

Yamada S, Ohtera H, Narihisa H (1986) Software reliability growth models with testing-effort. IEEE Trans Reliab 35(1):19–23

Yamada S, Hishitani J, Osaki S (1991) Test-effort dependent software reliability measurement. Int J Syst Sci 22(1):73–83

Yamada S, Tokuno K, Osaki S (1992) Imperfect debugging models with fault introduction rate for software reliability assessment. Int J Syst Sci 23(12):2241–2252

Yamada S, Tokuno K, Kasano Y (1998) Quantitative assessment models for software safety/reliability. Electron Commun Jpn (Part II: Electron) 81(5):33–43

Yang J, Liu Y, Xie M, Zhao M (2016) Modeling and analysis of reliability of multi-release open source software incorporating both fault detection and correction processes. J Syst Softw 115:102–110

Yazdanbakhsh O, Dick S, Reay I, Mace E (2016) On deterministic chaos in software reliability growth models. Appl Soft Comput 49:1256–1269

Zhang X, Pham H (2000) An analysis of factors affecting software reliability. J Syst Softw 50(1):43–56

Zhang X, Shin MY, Pham H (2001) Exploratory analysis of environmental factors for enhancing the software reliability assessment. J Syst Softw 57(1):73–78

Zhang X, Teng X, Pham H (2003) Considering fault removal efficiency in software reliability assessment. IEEE Trans Syst Man Cybern Part A Syst Hum 33(1):114–120

Zhu M, Pham H (2016) A software reliability model with time-dependent fault detection and fault removal. Vietnam J Comput Sci 3(2):71–79

Zhu M, Pham H (2017a) A two-phase software reliability modeling involving with software fault dependency and imperfect fault removal. Comput Lang Syst Struct 53:27–42

Zhu M, Pham H (2017b) A multi-release software reliability modeling for open source software incorporating dependent fault detection process. Ann Oper Res. https://doi.org/10.1007/s10479-017-2556-6

Zhu M, Pham H (2017c) Environmental factors analysis and comparison affecting software reliability in development of multi-release software. J Syst Softw 132:72–84

Zhu M, Pham H (2018) A Software reliability model incorporating martingale process with gamma-distributed environmental factors. Ann Oper Res. https://doi.org/10.1007/s10479-018-2951-7

Zhu M, Pham H (2020) A generalized multiple environmental factors software reliability model with stochastic fault detection process. Ann Oper Res. https://doi.org/10.1007/s10479-020-037 32-3

Zhu M, Zhang X, Pham H (2015) A comparison analysis of environmental factors affecting software reliability. J Syst Softw 109:150–160