

# Chapter 2

## Background: Software Quality and Reliability Prediction

### 2.1 Introduction

Size, complexity, and human dependency on software-based products have grown dramatically during past decades. Software developers are struggling to deliver reliable software with acceptable level of quality, within given budget and schedule. One measure of software quality and reliability is the number of residual faults. Therefore, researchers are focusing on the identification of the number of fault presents in the software or identification of program modules that are most likely to contain faults. A lot of models have been developed using various techniques. A common approach is followed for software reliability prediction utilizing failure data. Software reliability and quality prediction is highly desired by the stakeholders, developers, managers, and end users. Detecting software faults early during development will definitely improve the reliability and quality in cost-effective way.

In this chapter, a review of the recent available literature on software reliability and quality prediction is presented. The following sections cover the literature surveys on software reliability models, reliability-relevant software metrics, software capability maturity models, software defect prediction model, and software quality prediction models, regression testing and test case prioritization, and operational profile-based testing.

### 2.2 Software Reliability Models

A number of analytical models have been presented in literature to address the problem of software reliability measurement. These approaches are based mainly on the failure history of software and can be classified according to the nature of the failure process. The various software reliability models may be categorized as failure rate model, fault count model, software reliability growth models, etc. Details about these models can be found in Musa et al. (1987), Goel and Okumoto (1979), Pham (2006), Lyu (1996). These reliability prediction models attempt to

predict the reliability of the software in the later stages of the life cycle (testing and beyond). However, the opportunity of controlling software development process for cost-effectiveness is missed.

Software reliability models usually refer to estimating the number of remaining errors in a partially debugged software. Tests performed on the software derive outcome as accepted, conditionally accepted, or rejected. Acceptance may be based on the number of errors found over a selected period of time, on the number of paths executed of the total number of paths available to be executed, or some other prearranged criterion. Variety of reliability models are now competing for the attention of the analyst. Software reliability models can be broadly categorized as suggested by Pham (2006):

- *Deterministic* used to study the number of distinct operators and operands as well as the machine instructions in the program. Two most common deterministic models are:
  - Halstead’s software metric, based on unique no. of operators and operands.
  - McCabe’s cyclomatic complexity metric, based on cyclomatic number  $V(G)$ .
- *Probabilistic* describes the failure occurrence and/or fault removal phenomenon of the testing process as probabilistic events with respect to time and/or testing effort. Some common probabilistic models include the following (Pham 2006):
  - Failure rate model (times between failure models).
  - Failure or fault count model (NHPP models).
  - Error or fault-seeding model.
  - Reliability growth model, etc.

### 2.2.1 Failure Rate Models

Failure rate models are one of the earliest classes of models proposed for software reliability assessment. The most common approach is to assume that the time between  $(i - 1)$  and the  $i$ th failures follows a distribution whose parameters depend on the number of faults remaining in the program during this interval. Estimates of the parameters are obtained from the observed values of times between failures or failure count in intervals. The estimates of software reliability, mean time to next failure, etc., are then obtained from the fitted model. The failure models can also be distinguished according to the nature of the relationship between the successive failures rates by Lyu (1996): (1) deterministic relationship, which is the case for most failure rate models and, (2) stochastic relationship. Following are the list of some failure rate models (Goel 1985):

- Jelinski and Moranda Model.
- Schick and Wolverton Model.
- Goel and Okumoto Imperfect Debugging Model, etc.

### ***2.2.2 Failure or Fault Count Models***

The interest of this class of models is in the number of faults or failures in specified time intervals rather than in times between failures. The failure counts are assumed to follow a known stochastic process with a time-dependent discrete or continuous failure rate. Parameters of the failure rate can be estimated from the observed values of failure counts or from failure times. Estimates of software reliability, mean time to next failure, etc., can again be obtained from the relevant equations. The key models in this class are Goel (1985) as follows:

- Shooman exponential model.
- Musa execution time model.
- Goel–Okumoto non-homogeneous Poisson process model (G–O–NHPP).
- S-shape growth model.
- Discrete reliability growth model.
- Musa–Okumoto logarithmic Poisson execution model.
- Generalized NHPP, etc.

The variety of existing NHPP models can be classified according to the several different classification systems (Kapur et al. 1990), and one important thing in this categorization is that they are not mutually disjoint. Some of these models are as follows:

- Modeling under perfect debugging environment.
- Modeling the imperfect debugging and error generation phenomenon.
- Modeling with testing effort.
- Testing domain-dependent software reliability modeling.
- Modeling with respect to testing coverage.
- Modeling the severity of faults.
- Software reliability modeling for distributed software systems.
- Modeling fault detection and correction with time lag.
- Managing reliability in operational phase.
- Software reliability assessment using SDE model.
- Neural network–based software reliability modeling, etc.

### ***2.2.3 Error or Fault-Seeding Models***

The basic approach in this class of models is to “seed” known number of faults in the program which is assumed to have an unknown number of indigenous faults. The program is tested and observed for numbers of seeded and indigenous faults. From these, an estimate of the indigenous fault content of the program is obtained and used to assess software reliability and other relevant measures. Mills’ error seeding model (Mills 1972) is one the most popular and basic model to estimate the number of error in a program by introducing seeded errors into the program.

They have estimated unknown number of inherent errors from debugging data which consists of inherent and seeded errors.

Lipow (1972) modified Mills' model by taking into consideration the probability of finding a fault, of either kind, in any test of the software. Cai (1998) modified Mills' model and estimated the number of faults remaining in the software. Tohma et al. (1991) proposed a model for estimating the number of faults initially resident in a program at the beginning of the test or debugging process based on the hypergeometric distribution.

### 2.2.4 Reliability Growth Models

A software reliability growth model is applicable during the testing phase of software development and quantifies the software reliability in terms of estimated number of software error remaining in software or estimated time intervals between software failures. Software reliability growth is defined as the phenomenon that the number of software errors remaining in the system decreases with the progress of testing. For software growth modeling and analysis, the calendar time or CPU time is often used as the unit of software error detection period. However, the appropriate unit of software error detection period is sometimes the number of test runs or the number of executed test cases. A software reliability growth model for such a case is called a *discrete software reliability growth model*.

Software reliability growth models have been grouped into two classes of models: concave and S-shaped. These two model types are shown in Fig. 2.1. The most important thing about both models is that they have the same asymptotic behavior, that is, the defect detection rate decreases as the number of defects detected (and repaired) increases, and the total number of defects detected approach a finite value asymptotically (Table 2.1).

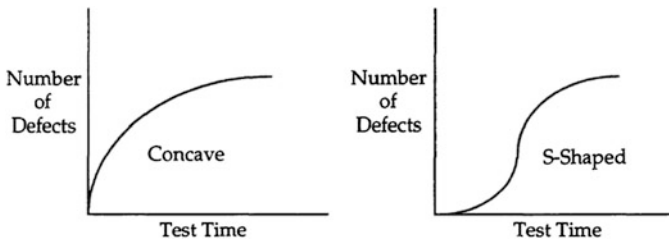


Fig. 2.1 Concave and S-shaped model

**Table 2.1** Software reliability growth models (Wood 1996)

Model name	Model type	$\mu$ (t)	Comments
Goel–Okumoto (G–O)	Concave	$a(1 - e^{-bt})$ $a \geq 0, b > 0$	Also called Musa model or exponential model
G–O S-shaped	S-shaped	$a(1 - (1 + bt)e^{-bt})$ $a \geq 0, b > 0$	Modification of G–O model to make it S-shaped
Hossain Dahiya	Concave	$a(1 - e^{-bt}) / (1 + ce^{-bt})$ $a \geq 0, b > 0, c > 0$	Becomes same as G–O as $c$ approaches to 0
Yamada exponential	Concave	$a(1 - e^{-r\alpha(1 - e^{-\beta t})})$ $a \geq 0, r\alpha > 0, \beta > 0$	Attempts to account for testing effort
Yamada Raleigh	S-shaped	$a(1 - e^{-r\alpha(1 - e^{-(\beta t^2/2)})})$ $a \geq 0, r\alpha > 0, \beta > 0$	Attempts to account for testing effort
Weibull	Concave	$a(1 - e^{-bt^c})$ $a \geq 0, b > 0, c > 0$	Same as G–O for $c = 1$
Kapur et al.	Concave	$a(1 - e^{-bpt})$	Based on imperfect debugging

## 2.3 Architecture-based Software Reliability Models

The software reliability prediction models which are based on number of fault/failures and times between failures are called black-box approach to predict software reliability. The common feature of black-box models is the stochastic modeling of the failure process, assuming some parametric model of cumulative number of failures over a finite time interval or of the time between failures (Gokhale et al. 2004). Variety of black-box reliability growth models can be found in the books and papers such as Musa et al. (1987), Pham (2006), Lyu (1996), Goel (1985).

The problem with these models is that the system is considered as a whole and only its interaction with external world is modeled without considering its internal architecture. Usually, in these models, no information is used except failure data for assessing or predicting software reliability. Therefore, these black-box models are not realistic to model a large component-based software system where various software modules are developed independently. So, a need of white-box approach is realized to estimate software reliability by considering the internal architecture of software.

Thus, the main goal of architecture-based software reliability model is to estimate the system reliability by considering the architecture of the software components and their interaction with other ones. Also, these models assume that components fail independently and that a component failure will ultimately lead to a system failure. Failure can happen during an execution period of any module or during the control transfer between two modules. The failure behavior of the modules and of the interfaces between the modules can be specified in terms of their reliabilities or failure rates. It is assumed that the transfer of control between

modules has a Markov property which means that given the knowledge of the module in control at any given time, the future behavior of the system is conditionally independent of the past behavior. Most of the architecture-based software reliability models are based on Markov process. A Markov process has the property that the future behavior of the process depends only on the current state and is independent of its past history. The various architecture-based software reliability model can be found in literatures such as Littlewood (1979), Popstojanova and Trivedi (2001), (2006), Gokhale et al. (2004), Gokhale (2007).

## 2.4 Bayesian Models

This group of models views reliability growth and prediction in a Bayesian framework rather than simply counting number of faults or failures. Failure and fault count model assumes that impact of each fault will be the same with respect to reliability. Also, these models allow change in the reliability only when failures occur. A Bayesian model takes a subjective viewpoint in that if no failures occur while the software is observed then the reliability should increase, reflecting the growing confidence in the software by the end user. The reliability is therefore a function of both the number of faults that have been detected and the amount of failure-free operation. This function is expressed in terms of a prior distribution representing the view from past data and a posterior distribution that incorporates past and current data.

The Bayesian models also reflect the belief that different faults have different impact on the reliability of the program. The number of faults is not as important as their impact. A program having number of faults in rarely used code will be more reliable than a program with only one fault in frequently used code. The Bayesian model says that it is more important to look at the behavior of the software than to estimate the number of faults in it. One of the very first models of this category is the Littlewood-Verrall reliability growth model (Littlewood and Verrall 1973).

## 2.5 Early Software Reliability Prediction Models

All the approaches discussed earlier for reliability prediction attempt to predict the reliability of the software in the later stages of the life cycle (testing and beyond). However, the opportunity of controlling software development process for cost-effectiveness is missed. Therefore, the need of early software reliability prediction is realized. Early reliability prediction attracts software professionals as it provides an opportunity for the early identification of software quality, cost overrun, and optimal development strategies. During the requirements, design, or coding phase,

predicting the number of faults can lead to mitigating actions such as additional reviews and more extensive testing.

Gaffney and Pietrolewicz (1990) have developed a phase-based model to predict reliability during test and operation using fault statistics obtained during the technical review of requirements, design, and the coding phase. One of the earliest and well-known studies to predict software reliability in the earlier phase of the life cycle is the work initiated by the Rome laboratory (1992). For their model, they developed prediction of fault density which they could then be transformed into other reliability measures such as failure rates. Li et al. (2003) proposed a framework based on expert opinion elicitation, developed to select the software engineering measures which are the best software reliability indicators. In a similar direction, Kumar and Misra (2008) made an effort for early software reliability prediction considering the six top-ranked measures given by Li et al. (2003) and software operational profile.

## 2.6 Reliability-Relevant Software Metrics

In order to achieve high software reliability, the number of faults in delivered code should be reduced. Furthermore, to achieve the target software reliability efficiently and effectively, it needs to be known at early stages of software development process. One way of knowing software reliability during early stages of development is early software reliability prediction. Since early phase of software life cycle testing/field failure data is not available, information available such as reliability-relevant software metrics, developer's maturity level, and expert opinions can be utilized to predict the number of faults in the software.

IEEE had developed a standard IEEE Std. 982.2 (1988) known as "IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software." The goal of the IEEE dictionary is to support software developers, project managers, and system users in achieving optimum reliability levels in software products. It was designed to address the needs of software developers and customers who are confronted with a surplus of models, techniques, and measures in the literature, but who lack sufficient guidance to utilize them effectively. The standard addresses the need for a uniform interpretation of these and other indicators of reliability. The IEEE dictionary assumes an intimate relationship between the reliability of a product and the process used to develop that product. The reliable product provides confirmation of a successful process; the unreliable product provides a lesson for process change. It is therefore the metrics selected for reliability assessment/prediction should provide insight into both process and product so that the essential facts necessary for process evaluation and required changes can be made effectively.

The measures are selected to provide information throughout the life cycle of a product. The basic goal is to provide the elements of a measurement program that support a constructive approach for achieving optimum reliability of the end

product. The selected measures are related to various product and process factors that may have an impact on software reliability and that are observable throughout the life cycle. The IEEE dictionary provides, even in the early life cycle phases, a means for continual assessment of the evolving product and the process. Through early reliability assessment, the IEEE dictionary supports constructive optimization of reliability within given performance, resource, and schedule constraints.

The dictionary focuses on measures of the potential causes of failure throughout the software life cycle, rather than measures affecting reliability of nearly completed products. The intent is to produce software that is reliable, rather than just an estimate of reliability nearly completed and possibly unreliable product. Both the traditional approach of measuring reliability and the constructive approach of building in reliability are placed in context in this dictionary. The primitives to be used and the method of computation are provided for each measure. The standard calibrates the rulers, the measurement tools, through the use of common units and a common language. It promotes the use of a common database in the industry. Through commonality, the standard provides the justification for building effective tools for the measurement process itself. The standard and this guide are intended to serve as the foundation on which researchers and practitioners can build consistent methods. These documents are designed to assist management in directing product development and support toward specific reliability goals. The purpose is to provide a common set of definitions through which a meaningful exchange of data and evaluations can occur. Successful application of the measures is dependent on their use as intended in the specified environments.

Fenton (1991) has classified software metrics into three broad categories: product, process, and resources metrics. Product metrics describe characteristics of the product such as size, complexity, design features, performance, and quality level. Process metrics can be used to improve software development process and maintenance. Resources metrics describe the project characteristics and execution. Several researchers (e.g., Zhang and Pham 2000; Li et al. 2003) have shown that approximately thirty software metrics can be associated with different phases of software development life cycle. Among these metrics, some are significant predictor to reliability. Zhang and Pham (2000) have presented the findings of empirical research from 13 companies participating in software development to identify the factors that may impact software reliability. They have shown that thirty-two potential factors are involved during the various phases of software development process.

## 2.7 Software Capability Maturity Models

The capability maturity model (CMM) has become a popular methodology to develop high-quality software within budget and time. The CMM framework includes 18 key process areas such as quality assurance, configuration management, defect prevention, peer review, and training. A software process is assigned



the highest maturity level if the practices in all 18 key process areas of the CMM are adopted. The CMM practices aid in reducing defect injection and in early identification of defects. As a consequence, the number of errors detected in testing and remaining in the delivered software will become lesser (Agrawal and Chari 2007). For example, as a software unit at Motorola improved from CMM level 2 to 5, the average defect density reduced from 890 defects per million assembly equivalent lines of code to about 126 defects per million assembly equivalent lines.

Paulk et al. (1993) provided an overview of the Capability Maturity Model for software development process. In their paper, they have discussed the software engineering and management practices that characterize organizations as they mature their processes for developing and maintaining software. Diaz et al. (1997) presented a case study that show average defect density reduced with increasing CMM level at Motorola. In a similar study, Krishnan and Kellner (1999) found that process maturity and personnel capability are significant predictors (both at the 10% level) of the number of defects. In an empirical study using 33 software products developed over 12 years by an IT company, Harter et al. (2000) found that 1% improvement in process maturity resulted in 1.589% increase in product quality. Agrawal et al. (2007) provided some results, which indicated that the biggest rewards from high levels of process maturity came from the reduction in variance of software development outcomes that were caused by factors other than software size.

## 2.8 Software Defects Prediction Models

Reliability of software system can be adversely affected by the number of residual faults present in the system. The main goal of software developers is to minimize the number of faults (defects) in the delivered code. An objective of all software projects is to minimize the number of residual faults in the delivered code and thus improving quality and reliability. Improving reliability is a key objective during system development and field deployment, and defect removal is the bottleneck in achieving this objective.

Lipow (1982) showed that the number of faults or “bugs” per line of code can be estimated based upon Halstead’s software science relationships. This number is shown to be an increasing function of the number of lines of code in a program, a result in agreement with intuition, and some current theories of complexity. A similar kind of work has been carried out by Yu et al. (1988), in which they presented the results by analyzing several defect models using data collected from two large commercial projects.

In another study, Levendel (1990) proposed a birth–death mathematical model based on different defect behavior. Paper assumed that defect removal is ruled by the “laws of the physics” of defect behavior that controls the defect removal process. The time to defect detection, the defect repair time, and the factor of introduction of new defects due to imperfect defect repair are some of the

“constants” in the laws governing defect removal. Test coverage is a measure of defect removal effectiveness. A model for projecting software defects from analyses of Ada design have been described by Agresti et al. (1992). The model predicted defect density based on the product and process characteristics. In a similar study, Wohlin et al. (1998) have presented two new methods to estimate the number of remaining defects after a review and hence control the software development process. The method is based on the review information from the individual reviewers and through statistical inferences. Conclusions about the remaining number of defects are then drawn after the reviews.

A critical review of software defect prediction model is provided by Fenton et al. (1999). This paper also proposes a model to improve the defect prediction situation by describing a prototype Bayesian belief network (BBN). A comprehensive evaluation of capture–recapture models for estimating software defect content was provided by Briand et al. (2000). Emam et al. (2001) have provided an extensive Monte Carlo simulation that evaluated capture–recapture models suitable for two inspectors assuming a code inspections context.

Fenton et al. (2008) presented a causal model for defect prediction using Bayesian nets. The main feature that distinguishes it from other defect prediction models is the fact that it explicitly combines both quantitative and qualitative factors. They have also presented a dataset for 31 software development projects. This dataset incorporates the set of quantitative and qualitative factors that were previously built into a causal model of the software process. The factors include values for code size, effort, and defects, together with qualitative data values judged by project managers using a questionnaire. Their model predicts the number of software defects that will be found in independent testing or operational usages with a satisfactorily predictive accuracy. Also, they have demonstrated that predictive accuracy increases with increasing project size. Catal et al. (2009) provided a systematic review of various software fault prediction studies with a specific focus on metrics, methods, and datasets.

Pandey and Goyal (2009) have developed an early fault prediction model using software metrics and process maturity which predicts the number of faults present before testing. The reliability of a software system depends on the number of residual faults sitting dormant inside. In fact, many of the software reliability models attempt to measure the number of residual bugs in the program (e.g., Briand et al. 2000; Emam et al. 2001; Fenton et al. 2008).

## 2.9 Software Quality Prediction Models

Assuring quality of large and complex software systems are challenging as these systems are developed by integrating various independent software modules. The quality of the system will depend on the quality of individual modules. All the modules are neither equally important nor do they contain an equal amount of

faults. Therefore, researchers started focusing on the classification of these modules as fault-prone and not fault-prone.

Software reliability and quality prediction model is of a great interest among the software quality researchers and industry professionals. Software quality prediction can be done by predicting the expected number of software faults in the modules (Khoshgoftaar Allen 1999; Khoshgoftaar and Seliya 2002) or classifying the software module as fault-prone (FP) or not fault-prone (NFP). A commonly used software quality classification model is to classify the software modules into one of the following two categories: FP and NFP. A lot of efforts have been made for FP module prediction using various methods such as classification tree (Khoshgoftaar and Seliya 2002), neural networks (Singh et al. 2008), support vector machine (Singh et al. 2009) fuzzy logic (Kumar 2009) and logistic regression (Schneidewind 2001).

On reviewing literature, it is found that supervised (Menzies et al. 2007), semi-supervised (Seliya and Khoshgoftaar 2007a), and unsupervised learning (Catal and Diri 2008) approaches have been used for building a fault prediction models. Among these, supervised learning approach is widely used and found to be more useful FP module prediction if sufficient amount of fault data from previous releases are available. Generally, these models use software metrics of earlier software releases and fault data collected during testing phase. The supervised learning approaches cannot build powerful models with limited data. Therefore, some researchers presented a semi-supervised classification approach (Seliya and Khoshgoftaar 2007a) for software fault prediction with limited fault data. Unsupervised learning approaches such as clustering methods can be used in the absence of fault data. In most cases, software metrics and fault data obtained from a similar project or system release previously developed are used to train a software quality model. Subsequently, the model is applied to program modules of software currently under development for classifying them into the FP and NFP groups.

Various classification models have been developed for classifying a software module as FP and NFP. Schneidewind (2001) utilizes logistic regression in combination with Boolean discriminant functions for predicting FP software modules. Khoshgoftaar and Seliya (2002) incorporated three different regression tree algorithms CART-LS, S-PLUS, and CART-LAD into a single case study to show their effectiveness in finding the number of faults predicted using them. A study conducted by Khoshgoftaar and Seliya (2003) compared the fault prediction accuracies of six commonly used prediction modeling techniques. The study conducted a large-scale case study consisting of data collected over four successive system releases of a very large legacy telecommunications system. Some other works that have focused on FP module prediction include Munson and Khoshgoftaar (1992), Ohlsson and Alberg (1996), El-Emam et al. (2001).

Pandey and Goyal (2009) have presented an approach for prediction of the number of faults present in any software using software metrics. They have shown that software metrics are good indicators of software quality, and the number of faults present in the software. In certain scenarios, prediction of exact number of

fault is not desirable and one needs to have the information about the quality of the software module. Software quality prediction can be done by predicting the number of software faults expected in the modules or classifying the software module as FP or NFP. Therefore, researchers started focusing on the classification of these modules as FP and NFP.

From the literature, it has been found that the decision tree induction algorithms such as CART, ID3, and C4.5 are efficient techniques for FP module classification. These algorithms use crisp value of software metrics and classify the module as a FP or NFP. It has been found that early-phase software metrics have fuzziness in nature and crisp value assignment seems to be impractical. Also, a software module cannot be completely FP or NFP. In other words, it is unfair to assign a crisp value of software module representing its fault proneness.

## 2.10 Regression Testing

Regression testing is an important and expensive software maintenance activity to assure the quality and reliability of modified software. To reduce the cost of regression testing, researches have proposed many techniques such as regression test selection (RTS), test suite minimization (TSM), and test case prioritization (TCP). A test suite minimization technique is given by Harrold et al. (1993) to select a representative set of test cases from a test suite providing the same coverage as the entire test suite. Rothermel and Harrold (1996) have presented a framework for evaluating regression test selection techniques that classifies techniques in terms of inclusiveness, precision, efficiency, and generality. Wong et al. (1997) have found that both TSM and TCP suffers from certain drawbacks in some situation and suggested test case prioritization according to the criterion of increasing cost per additional coverage. Later, Rothermel et al. (1999) presented several techniques for prioritizing test cases and they empirically evaluated their ability to improve rate of fault detection—a measure of how quickly faults are detected within the testing process. For this, they provided a metric, APFD, which measures the average cumulative percentage of faults detected over the course of executing the test cases in a test suite in a given order. Their results have shown that test case prioritization can significantly improve the rate of fault detection of test suites. As a result of this, efforts have been made by many researchers on test case prioritization in order to improve fault detection rate (Elbaum et al. 2000, 2002, 2003, 2004; Do et al. 2006; Qu et al. 2007; Park et al. 2008; Khan et al. 2009; Kim and Baik 2010).

Review of literature indicate that earlier test case prioritization techniques have not considered the factors, such as program change level (PCL), test suite change level (TCL), and test suite size (TS) that affect the cost-effectiveness of the prioritization techniques. Also, all the traditional test case techniques presented to date have used a straightforward prioritization approach using some coverage criteria. We have found these traditional techniques are based on coverage

information which relies on data gathered on the original version of a program (prior to modifications) in their prioritizations. They have ignored the information from the modified program version that may definitely affect the cost of prioritization.

## 2.11 Operational Profile

Reliability is a user-oriented view and strongly tied to the operational usage of the product. Operational profile becomes particularly valuable in guiding test planning by assigning test cases to different operations in accordance with their probabilities of occurrence. In the case of software, the operational usage information can be used to develop the various profiles such as customer profile, user profile, system mode profile, functional profile, and operational profile (Musa 2005).

One of the pioneer researches about the development of operational profile is by John D. Musa from AT&T Bell Laboratories (Musa 1993). It is a practical approach to ensure that a system is delivered with a maximum reliability, because the operations most frequently used also get tested the most. Musa informally characterized the benefits-to-cost ratio as 10 or greater (Musa 2005). In 1993, AT&T had used an operational profile successfully for the testing of a telephone switching service, which significantly reduced the number of problems reported by customers (Koziolk 2005). Also, Hewlett-Packard reorganized its test processes with operational profiles and reduced testing time and cost for a multiprocessor operating system by 50% (Koziolk 2005). Arora et al. (2005) conducted a case study on Pocket PC, a Windows CE 3.0-based device, and demonstrated a significant reliability improvement through operational profile-driven testing. It has been observed by many researchers and industries professional that operational profile-based testing is useful when the estimates of test cases are available (based on the constraints of the testing resource and time). Recently, Pandey et al. (2012) has presented a model-based approach to optimize the validation efforts by integrating the functional complexity and operational profile of fog light ECUs of an automotive system.

## 2.12 Observations

On reviewing literatures, the following observations have emerged:

1. Failure data are not available in the early phases of software life cycle and the information such as reliability-relevant software metrics, developer's maturity level, and expert opinions can be used. Both software metrics and process maturity play a vital role in early fault prediction in the absence of failure data.

Therefore, integrating software metrics with process maturity will provide a better fault prediction accuracy.

2. Early fault prediction is useful for both software engineers and managers since it provides vital information for making design and resource allocation decisions and thereby facilitates efficient and effective development process. Therefore, a model to predict number of faults present at the end of each phase of software life cycle is required. An early fault prediction model using software metrics process maturity is discussed in the [Chap. 3](#).
3. One of the measures of software reliability is the number of residual faults and system reliability will be lesser as the number of residual defects (faults) in the system becomes more. There are several faults prediction models, but predicting faults without field failure data before testing are rarely discussed. Therefore, there is a need of a fault prediction model which predicts number of residual faults which may likely to occur after testing or operational use. Considering the importance of residual faults in reliability prediction, a multistage model for residual fault prediction is discussed in the [Chap. 4](#).
4. Prediction of exact number of fault in a software module is not always necessary and there must be some measure of categorization which classify software module as FP or NFP. This will definitely help to improve the reliability and quality of software products by better resource utilization during software development process. Therefore, a model for prediction and ranking of FP software module is presented and discussed in the [Chap. 5](#).
5. Regression testing is vital for reliability assurance of a modified program. It is one of the most expensive testings. Considering these points, a cost-effective reliability centric test case prioritization approach is parented in the [Chap. 6](#).
6. The reliability of software, much more so than the reliability of hardware, is strongly tied to the operational usage of an application. Making a good reliability estimate of software depends on testing the product as per its field usages. Considering these points, reliability centric operational profile-based testing approach is discussed in the [Chap. 7](#).

## References

- Musa, J. D., Iannino, A., & Okumoto, K. (1987). *Software reliability: measurement, prediction, and application*. New York: McGraw-Hill Publication.
- Goel, A. L., & Okumoto, K. (1979). A time-dependent error detection rate model for software reliability and other performance measure. *IEEE Transaction on Reliability*, R-28, 206–211.
- Pham, H. (2006). *System software reliability, reliability engineering series*. London: Springer.
- Lyu, M. R. (1996). *Handbook of software reliability engineering*. NY: McGraw-Hill/IEE Computer Society Press.
- Goel, A. L. (1985). Software reliability models: assumptions, limitations, and applicability. *IEEE Transaction on Software Engineering*, SE-11(12), 1411–1423.

- Kapur, P. K., & Garg, R. B. (1990). A software reliability growth model under imperfect debugging. *RAIRO*, 24, 295–305.
- Mills, H. D. (1972). *On the statistical validation of computer program* (pp. 72–6015). Gaithersburg, MD: IBM Federal Systems Division.
- Lipow, M. (1972). *Estimation of software package residual errors*. Software Series Report TRW-SS-09, Redondo Beach, CA: TRW.
- Cai, K. Y. (1998). On estimating the number of defects remaining in software. *Journal of System and Software*, 40(1).
- Tohma, Y., Yamano, H., Ohba, M., & Jacoby, R. (1991). The estimation of parameter of the hypergeometric distribution and its application to the software reliability growth model. *IEEE Transaction on Software Engineering*, SE 17(2).
- Wood, A. (1996). *Software reliability growth models*. Technical report 96.1, part number 130056.
- Gokhale, S. S., Wong, W. E., Horgan, J. R., & Trivedi, K. S. (2004). An analytical approach to architecture-based software performance and reliability prediction. *Performance Evaluation*, 58, 391–412.
- Littlewood, B. (1979). Software reliability model for modular program structure. *IEEE Transaction on Reliability*, R-28(3), 241–247.
- Popstojanova, K. G., & Trivedi, K. S. (2001). Architecture-based approach to reliability assessment of software systems. *Performance Evaluation*, 45, 179–204.
- Gokhale, S. S., & Trivedi, K. S. (2006). Analytical models for architecture-based software reliability prediction: a unification framework. *IEEE Transaction on Reliability*, 55(4), 578–590.
- Gokhale, S. S. (2007). Architecture-based software reliability analysis: overview and limitations. *IEEE Transaction on Dependable and Secure Computing*, 4(1), 32–40.
- Littlewood, B., & Verrall, J. (1973). A bayesian reliability growth model for computer software. *Journal of the Royal Statistical Society, series C*, 22(3), 332–346.
- Gaffney, G. E., & Pietrolewicz, J. (1990). An automated model for software early error prediction (SWEEP). In *Proceeding of 13th Minnow Brook Workshop on Software Reliability*.
- Rome Laboratory (1992). *Methodology for software reliability prediction and assessment* (Vols. 1–2). Technical report RL-TR-92-52.
- Li, M., & Smids, C. (2003). A ranking of software engineering measures based on expert opinion. *IEEE Transaction on Software Engineering*, 29(9), 24–811.
- Kumar, K. S., & Misra, R. B. (2008). An enhanced model for early software reliability prediction using software engineering metrics. In *Proceedings of 2nd International Conference on Secure System Integration and Reliability Improvement* (pp. 177–178).
- IEEE (1988). IEEE guide for the use of IEEE standard dictionary of measures to produce reliable software. *IEEE Standard 982.2*.
- Fenton, N. (1991). *Software metrics: A rigorous approach*. London: Chapman & Hall.
- Zhang, X., & Pham, H. (2000). An analysis of factors affecting software reliability. *The Journal of Systems and Software*, 50(1), 43–56.
- Agrawal, M., & Chari, K. (2007). Software effort, quality and cycle time: A study of CMM level 5 projects. *IEEE Transaction on Software Engineering*, 33(3), 145–156.
- Paulk, M. C., Weber, C. V., Curtis, B., & Chrissis, M. B. (1993). Capability maturity model version 1.1. *IEEE Software*, 10(3), 18–27.
- Diaz, M., & Sligo, J. (1997). How software process improvement helped Motorola. *IEEE Software*, 14(5), 75–81.
- Krishnan, M. S., & Kellner, M. I. (1999). Measuring process consistency: implications reducing software defects. *IEEE Transaction on Software Engineering*, 25(6), 800–815.
- Harter, D. E., Krishnan, M. S., & Slaughter, S. A. (2000). Effects of process maturity on quality, cycle time and effort in software product development. *Management Science*, 46, 451–466.
- Lipow, M. (1982). Number of faults per line of code. *IEEE Transaction on Software Engineering*, SE-8(4), 437–439.
- Yu, T. J., Shen, V. Y., & Dunsmore, H. E. (1988). An analysis of several software defect models. *IEEE Transaction on Software Engineering*, 14(9), 261–270.

- Levendel, Y. (1990). Reliability analysis of large software systems: Defect data modeling. *IEEE Transaction on Software Engineering*, 16(2), 141–152.
- Agresti, W. W., & Evanco, W. M. (1992). Projecting software defect from analyzing Ada design. *IEEE Transaction on Software Engineering*, 18(11), 988–997.
- Wohlin, C. & Runeson, P. (1998). Defect content estimations from review data. In *Proceedings of 20th International Conference on Software Engineering* (pp. 400–409).
- Fenton, N. E., & Neil, M. (1999). A critique of software defect prediction models. *IEEE Transaction on Software Engineering*, 25(5), 675–689.
- Briand, L. C., Emam, K. E., Freimut, B. G., & Laitenberger, O. (2000). A comprehensive evaluation of capture: Recapture models for estimating software defect content. *IEEE Transaction on Software Engineering*, 26(8), 518–540.
- El-Emam, K., Melo, W., & Machado, J. C. (2001). The prediction of faulty classes using object-oriented design metrics. *Journal of Systems and Software*, 56(1), 63–75.
- Fenton, N., Neil, N., Marsh, W., Hearty, P., Radlinski, L., & Krause, P. (2008). On the effectiveness of early life cycle defect prediction with Bayesian Nets. *Empirical of Software Engineering*, 13, 499–537.
- Catal, C., & Diri, B. (2009). Investigating the effect of dataset size, metrics set, and feature selection techniques on software fault prediction problem. *Information Sciences*, 179(8), 1040–1058.
- Pandey, A. K., & Goyal, N. K. (2009). A fuzzy model for early software fault prediction using process maturity and software metrics. *International Journal of Electronics Engineering*, 1(2), 239–245.
- Khoshgoftaar, T. M., & Allen, E. B. (1999). A comparative study of ordering and classification of fault-prone software modules. *Empirical Software Engineering*, 4, 159–186.
- Khoshgoftaar, T. M., & Seliya, N. (2002). Tree-based software quality models for fault prediction. In *Proceedings of 8th International Software Metrics Symposium, Ottawa, Ontario, Canada* (203–214).
- Khoshgoftaar, T. M., & Seliya, N. (2003). Fault prediction modeling for software quality estimation: comparing commonly used techniques. *Empirical Software Engineering*, 8, 255–283.
- Singh, Y., Kaur, A., & Malhotra, R. (2008). *Predicting software fault proneness model using neural network*. LNBIP 9, Springer.
- Singh, Y., Kaur, A., & Malhotra, R. (2009). Software fault proneness prediction using support vector machines. In *The Proceedings of the World Congress on Engineering, London, UK*, 1–3 July.
- Kumar, K. S. (2009). Early software reliability and quality prediction (Ph.D. Thesis, IIT Kharagpur, Kharagpur, India).
- Schneidewind, N. F. (2001). Investigation of logistic regression as a discriminant of software quality. In *The Proceedings of 7th International Software Metrics Symposium, London, UK* (pp. 328–337).
- Menzies, T., Greenwald, J., & Frank, A. (2007). Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1), 2–13.
- Seliya, N., & Khoshgoftaar, T. M. (2007). Software quality estimation with limited fault data: A semi-supervised learning perspective. *Software Quality Journal*, 15(3), 327–344.
- Catal, C., & Diri, B. (2008). A fault prediction model with limited fault data to improve test process. In *Proceedings of the 9th International Conference on Product Focused Software Process Improvement* (pp. 244–257).
- Munson, J. C., & Khoshgoftaar, T. M. (1992). The detection of fault-prone programs. *IEEE Transactions on Software Engineering*, 18(5), 423–433.
- Ohlsson, N., & Alberg, H. (1996). Predicting fault-prone software modules in telephone switches. *IEEE Transaction on Software Engineering*, 22(12), 886–894.
- Harrold, M., Gupta, R., & Soffa, M. (1993). A methodology for controlling the size of a test suite. *ACM Transaction on Software Engineering and Methodology*, 2(3), 270–285.



- Rothermel, G., & Harrold, M. J. (1996). Analyzing regression test selection techniques. *IEEE Transaction on Software Engineering*, 22(8), 529–551.
- Wong, W. E., Horgan, J. R., London, S. & Agrawal, H. (1997). A study of effective regression testing in practice. In *Proceedings of the Eighth Int'l Symposium on Software Reliability Engineering* (pp. 230–238).
- Rothermel, G., Untch, R. H., Chu, C., & Harrold, M. J. (1999). Test case prioritization: An empirical study. In *Proceedings of the Int'l Conf. Software Maintenance* (pp. 179–188).
- Elbaum, S., Malishevsky, A., & Rothermel, G. (2000). Prioritizing test cases for regression testing. In *Proceedings of the International Symposium on Software Testing and Analysis* (pp. 102–112).
- Elbaum, S., Malishevsky, A., & Rothermel, G. (2002). Test case prioritization: a family of empirical studies. *IEEE Transaction of Software Engineering*, 28(2), 159–182.
- Elbaum, S., Kallakuri, P., Malishevsky, A., Rothermel, G., & Kanduri, S. (2003). Understanding the effects of changes on the cost-effectiveness of regression testing techniques. *Journal of Software, Verification and Reliability*, 12(2), 65–83.
- Elbaum, S., Rothermel, G., Kanduri, S., & Malishevsky, A. G. (2004). Selecting a cost-effective test case prioritization technique. *Software Quality Journal*, 12(3), 185–210.
- Do, H., Rothermel, G., & Kinneer, A. (2006). Prioritizing Junit test cases: An empirical assessment and cost-benefits analysis. *Empirical Software Engineering*, 11, 33–70.
- Qu, B., Nie, C., Xu, B. & Zhang, X. (2007). Test case prioritization for black box testing. In *The Proceedings of 31st Annual International Computer Software and Applications Conference*.
- Park, H., Ryu, H., & Baik, J. (2008). Historical value-based approach for cost-cognizant test case prioritization to improve the effectiveness of regression testing. In *The Proceedings 2nd International Conference on Secure System Integration and Reliability Improvement* (pp. 39–46).
- Khan, S. R., Rehman, I., & Malik, S. (2009). The impact of test case reduction and prioritization on software testing effectiveness. In *Proceeding of International Conference on Emerging Technologies* (pp. 416–421).
- Kim, S., & Baik J. (2010). An effective fault aware test case prioritization by incorporating a fault localization technique. In *Proceedings of ESEM-10, Bolzano-Bozen, Italy* (pp. 16–17).
- Musa, J. D. (2005). *Software reliability engineering: more reliable software faster and cheaper* (2nd ed.). Tata McGraw-Hill Publication.
- Musa, J. D. (1993). Operational profiles in software reliability engineering. *IEEE Software Magazine*.
- Kozirolek, H. (2005). Operational profiles for software reliability. Seminar on Dependability Engineering, Germany.
- Arora, S., Misra, R. B., & Kumre, V. M. (2005). Software reliability improvement through operational profile driven testing. In *Proceedings of Annual IEEE Conference on Reliability and Maintainability Symposium, Virginia* (pp. 621–627).
- Pandey, A. K., Smith, J., & Diwanji, V. (2012). Cost effective reliability centric validation model for automotive ECUs. In *The 23rd IEEE International Symposium on Software Reliability Engineering, Dallas, TX USA* (pp. 38–44).



<http://www.springer.com/978-81-322-1175-4>

Early Software Reliability Prediction

A Fuzzy Logic Approach

Pandey, A.K.; Goyal, N.K.

2013, XIX, 153 p., Hardcover

ISBN: 978-81-322-1175-4