Software Reliability Engineering: A Roadmap

Michael R. Lyu

Michael R. Lyu received the Ph.D. in computer science from University of California, Los Angeles in 1988. He is a Professor in the Computer Science and Engineering Department of the Chinese University of Hong Kong. He worked at the Jet Propulsion Laboratory, Bellcore, and Bell Labs; and taught at the University of Iowa. He has participated in more than 30 industrial projects, published over 250 papers, and helped to develop many commercial systems and software tools. Professor Lyu is frequently invited as a keynote or tutorial speaker to conferences and workshops in U.S., Europe, and Asia. He initiated the International Symposium on Software Reliability Engineering (ISSRE) in 1990. He also received Best Paper Awards in ISSRE'98 and in ISSRE'2003. Professor Lyu is an IEEE Fellow and an AAAS Fellow, for his contributions to software reliability engineering and software fault tolerance.

Future of Software Engineering(FOSE'07)
0-7695-2829-5/07 $20.00 © 2007 IEEE

Software Reliability Engineering: A Roadmap

Michael R. Lyu
Computer Science and Engineering Department
The Chinese University of Hong Kong, Hong Kong
lyu@cse.cuhk.edu.hk

Abstract

Software reliability engineering is focused on engineering techniques for developing and maintaining software systems whose reliability can be quantitatively evaluated. In order to estimate as well as to predict the reliability of software systems, failure data need to be properly measured by various means during software development and operational phases. Moreover, credible software reliability models are required to track underlying software failure processes for accurate reliability analysis and forecasting. Although software reliability has remained an active research subject over the past 35 years, challenges and open questions still exist. In particular, vital future goals include the development of new software reliability engineering paradigms that take software architectures, testing techniques, and software failure manifestation mechanisms into consideration. In this paper, we review the history of software reliability engineering, the current trends and existing problems, and specific difficulties. Possible future directions and promising research subjects in software reliability engineering are also addressed.

1. Introduction

Software permeates our daily life. There is probably no other human-made material which is more omnipresent than software in our modern society. It has become a crucial part of many aspects of society: home appliances, telecommunications, automobiles, airplanes, shopping, auditing, web teaching, personal entertainment, and so on. In particular, science and technology demand high-quality software for making improvements and breakthroughs.

The size and complexity of software systems have grown dramatically during the past few decades, and the trend will certainly continue in the future. The data from industry show that the size of the software for various systems and applications has been growing exponentially for the past 40 years [20]. The trend of such growth in the telecommunication, business, defense, and transportation industries shows a compound growth rate of ten times every five years. Because of this ever-increasing dependency, software failures can lead to serious, even fatal, consequences in safety-critical systems as well as in normal business. Previous software failures have impaired several high-visibility programs and have led to loss of business [28].

The ubiquitous software is also invisible, and its invisible nature makes it both beneficial and harmful. From the positive side, systems around us work seamlessly thanks to the smooth and swift execution of software. From the negative side, we often do not know when, where and how software ever has failed, or will fail. Consequently, while reliability engineering for hardware and physical systems continuously improves, reliability engineering for software does not really live up to our expectation over the years.

This situation is frustrating as well as encouraging. It is frustrating because the software crisis identified as early as the 1960s still stubbornly stays with us, and "software engineering" has not fully evolved into a real engineering discipline. Human judgments and subjective favorites, instead of physical laws and rigorous procedures, dominate many decision making processes in software engineering. The situation is particularly critical in software reliability engineering. Reliability is probably the most important factor to claim for any engineering discipline, as it quantitatively measures quality, and the quantity can be properly engineered. Yet software reliability engineering, as elaborated in later sections, is not yet fully delivering its promise. Nevertheless, there is an encouraging aspect to this situation. The demands on, techniques of, and enhancements to software are continually increasing, and so is the need to understand

Future of Software Engineering(FOSE'07)
0-7695-2829-5/07 $20.00 © 2007 IEEE

its reliability. The unsettled software crisis poses tremendous opportunities for software engineering researchers as well as practitioners. The ability to manage quality software production is not only a necessity, but also a key distinguishing factor in maintaining a competitive advantage for modern businesses.

Software reliability engineering is centered on a key attribute, software reliability, which is defined as the probability of failure-free software operation for a specified period of time in a specified environment [2]. Among other attributes of software quality such as functionality, usability, capability, and maintainability, etc., software reliability is generally accepted as the major factor in software quality since it quantifies software failures, which can make a powerful system inoperative. Software reliability engineering (SRE) is therefore defined as the quantitative study of the operational behavior of software-based systems with respect to user requirements concerning reliability. As a proven technique, SRE has been adopted either as standard or as best current practice by more than 50 organizations in their software projects and reports [33], including AT&T, Lucent, IBM, NASA, Microsoft, and many others in Europe, Asia, and North America. However, this number is still relatively small compared to the large amount of software producers in the world.

Existing SRE techniques suffer from a number of weaknesses. First of all, current SRE techniques collect the failure data during integration testing or system testing phases. Failure data collected during the late testing phase may be too late for fundamental design changes. Secondly, the failure data collected in the in-house testing may be limited, and they may not represent failures that would be uncovered under actual operational environment. This is especially true for high-quality software systems which require extensive and wide-ranging testing. The reliability estimation and prediction using the restricted testing data may cause accuracy problems. Thirdly, current SRE techniques or modeling methods are based on some unrealistic assumptions that make the reliability estimation too optimistic relative to real situations. Of course, the existing software reliability models have had their successes; but every model can find successful cases to justify its existence. Without cross-industry validation, the modeling exercise may become merely of intellectual interest and would not be widely adopted in industry. Thus, although SRE has been around for a while, credible software reliability techniques are still urgently needed, particularly for modern software systems [24].

In the following sections we will discuss the past, the present, and the future of software reliability engineering. We first survey what techniques have been proposed and applied in the past, and then describe what the current trend is and what problems and concerns remain. Finally, we propose the possible future directions in software reliability engineering.

2. Historical software reliability engineering techniques

In the literature a number of techniques have been proposed to attack the software reliability engineering problems based on software fault lifecycle. We discuss these techniques, and focus on two of them.

2.1. Fault lifecycle techniques

Achieving highly reliable software from the customer's perspective is a demanding job for all software engineers and reliability engineers. [28] summarizes the following four technical areas which are applicable to achieving reliable software systems, and they can also be regarded as four fault lifecycle techniques:

1) Fault prevention: to avoid, by construction, fault occurrences.

2) Fault removal: to detect, by verification and validation, the existence of faults and eliminate them.

3) Fault tolerance: to provide, by redundancy, service complying with the specification in spite of faults having occurred or occurring.

4) Fault/failure forecasting: to estimate, by evaluation, the presence of faults and the occurrences and consequences of failures. This has been the main focus of software reliability modeling.

Fault prevention is the initial defensive mechanism against unreliability. A fault which is never created costs nothing to fix. Fault prevention is therefore the inherent objective of every software engineering methodology. General approaches include formal methods in requirement specifications and program verifications, early user interaction and refinement of the requirements, disciplined and tool-assisted software design methods, enforced programming principles and environments, and systematic techniques for software reuse. Formalization of software engineering processes with mathematically specified languages and tools is an aggressive approach to rigorous engineering of software systems. When applied successfully, it can completely prevent faults. Unfortunately, its application scope has been

limited. Software reuse, on the other hand, finds a wider range of applications in industry, and there is empirical evidence for its effectiveness in fault prevention. However, software reuse without proper certification could lead to disaster. The explosion of the Ariane 5 rocket, among others, is a classic example where seemly harmless software reuse failed miserably, in which critical software faults slipped through all the testing and verification procedures, and where a system went terribly wrong only during complicated real-life operations.

Fault prevention mechanisms cannot guarantee avoidance of all software faults. When faults are injected into the software, fault removal is the next protective means. Two practical approaches for fault removal are software testing and software inspection, both of which have become standard industry practices in quality assurance. Directions in software testing techniques are addressed in [4] in detail.

When inherent faults remain undetected through the testing and inspection processes, they will stay with the software when it is released into the field. Fault tolerance is the last defending line in preventing faults from manifesting themselves as system failures. Fault tolerance is the survival attribute of software systems in terms of their ability to deliver continuous service to the customers. Software fault tolerance techniques enable software systems to (1) prevent dormant software faults from becoming active, such as defensive programming to check for input and output conditions and forbid illegal operations; (2) contain the manifested software errors within a confined boundary without further propagation, such as exception handling routines to treat unsuccessful operations; (3) recover software operations from erroneous conditions, such as checkpointing and rollback mechanisms; and (4) tolerate system-level faults methodically, such as employing design diversity in the software development.

Finally if software failures are destined to occur, it is critical to estimate and predict them. Fault/failure forecasting involves formulation of the fault/failure relationship, an understanding of the operational environment, the establishment of software reliability models, developing procedures and mechanisms for software reliability measurement, and analyzing and evaluating the measurement results. The ability to determine software reliability not only gives us guidance about software quality and when to stop testing, but also provides information for software maintenance needs. It can facilitate the validity of software warranty when reliability of software has been properly certified. The concept of scheduled maintenance with software rejuvenation techniques [46] can also be solidified.

The subjects of fault prevention and fault removal have been discussed thoroughly by other articles in this issue. We focus our discussion on issues related to techniques on fault tolerance and fault/failure forecasting.

2.2. Software reliability models and measurement

As a major task of fault/failure forecasting, software reliability modeling has attracted much research attention in estimation (measuring the current state) as well as prediction (assessing the future state) of the reliability of a software system. A software reliability model specifies the form of a random process that describes the behavior of software failures with respect to time. A historical review as well as an application perspective of software reliability models can be found in [7, 28]. There are three main reliability modeling approaches: the error seeding and tagging approach, the data domain approach, and the time domain approach, which is considered to be the most popular one. The basic principle of time domain software reliability modeling is to perform curve fitting of observed time-based failure data by a pre-specified model formula, such that the model can be parameterized with statistical techniques (such as the Least Square or Maximum Likelihood methods). The model can then provide estimation of existing reliability or prediction of future reliability by extrapolation techniques. Software reliability models usually make a number of common assumptions, as follows. (1) The operation environment where the reliability is to be measured is the same as the testing environment in which the reliability model has been parameterized. (2) Once a failure occurs, the fault which causes the failure is immediately removed. (3) The fault removal process will not introduce new faults. (4) The number of faults inherent in the software and the way these faults manifest themselves to cause failures follow, at least in a statistical sense, certain mathematical formulae. Since the number of faults (as well as the failure rate) of the software system reduces when the testing progresses, resulting in growth of reliability, these models are often called software reliability growth models (SRGMs).

Since Jelinsky and Moranda proposed the first SRGM [23] in 1972, numerous SRGMs have been proposed in the past 35 years, such as exponential failure time class models, Weibull and Gamma failure

time class models, infinite failure category models, Bayesian models, and so on [28, 36, 50]. Unified modeling approaches have also been attempted [19]. As mentioned before, the major challenges of these models do not lie in their technical soundness, but their validity and applicability in real world projects.

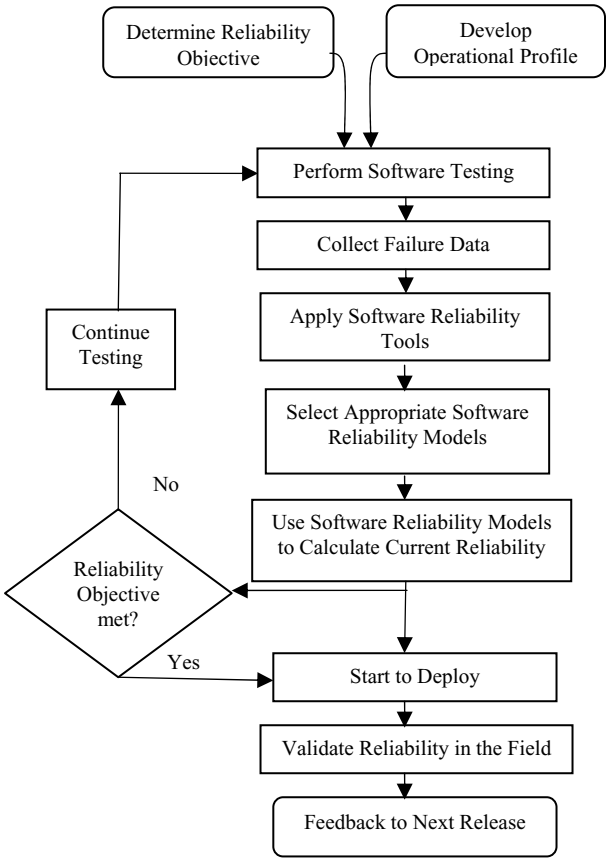

Figure 1. Software Reliability Engineering Process Overview

Figure 1 shows an SRE framework in current practice [28]. First, a reliability objective is determined quantitatively from the customer's viewpoint to maximize customer satisfaction, and customer usage is defined by developing an operational profile. The software is then tested according to the operational profile, failure data collected, and reliability tracked during testing to determine the product release time. This activity may be repeated until a certain reliability level has been achieved. Reliability is also validated in the field to evaluate the reliability engineering efforts and to achieve future product and process improvements.

It can be seen from Figure 1 that there are four major components in this SRE process, namely (1) reliability objective, (2) operational profile, (3) reliability modeling and measurement, and (4) reliability validation. A reliability objective is the specification of the reliability goal of a product from the customer viewpoint. If a reliability objective has been specified by the customer, that reliability objective should be used. Otherwise, we can select the reliability measure which is the most intuitive and easily understood, and then determine the customer's "tolerance threshold" for system failures in terms of this reliability measure.

The operational profile is a set of disjoint alternatives of system operational scenarios and their associated probabilities of occurrence. The construction of an operational profile encourages testers to select test cases according to the system's likely operational usage, which contributes to more accurate estimation of software reliability in the field.

Reliability modeling is an essential element of the reliability estimation process. It determines whether a product meets its reliability objective and is ready for release. One or more reliability models are employed to calculate, from failure data collected during system testing, various estimates of a product's reliability as a function of test time. Several interdependent estimates can be obtained to make equivalent statements about a product's reliability. These reliability estimates can provide the following information, which is useful for product quality management: (1) The reliability of the product at the end of system testing. (2) The amount of (additional) test time required to reach the product's reliability objective. (3) The reliability growth as a result of testing (e.g., the ratio of the value of the failure intensity at the start of testing to the value at the end of testing). (4) The predicted reliability beyond the system testing, such as the product's reliability in the field.

Despite the existence of a large number of models, the problem of model selection and application is manageable, as there are guidelines and statistical methods for selecting an appropriate model for each application. Furthermore, experience has shown that it is sufficient to consider only a dozen models, particularly when they are already implemented in software tools [28].

Using these statistical methods, "best" estimates of reliability are obtained during testing. These estimates are then used to project the reliability during field operation in order to determine whether the reliability objective has been met. This procedure is an iterative process, since more testing will be needed if the objective is not met. When the operational profile is not fully developed, the application of a test

compression factor can assist in estimating field reliability. A test compression factor is defined as the ratio of execution time required in the operational phase to execution time required in the test phase to cover the input space of the program. Since testers during testing are quickly searching through the input space for both normal and difficult execution conditions, while users during operation only execute the software with a regular pace, this factor represents the reduction of failure rate (or increase in reliability) during operation with respect to that observed during testing.

Finally, the projected field reliability has to be validated by comparing it with the observed field reliability. This validation not only establishes benchmarks and confidence levels of the reliability estimates, but also provides feedback to the SRE process for continuous improvement and better parameter tuning. When feedback is provided, SRE process enhancement comes naturally: the model validity is established, the growth of reliability is determined, and the test compression factor is refined.

2.3. Software fault tolerance techniques and models

Fault tolerance, when applicable, is one of the major approaches to achieve highly reliable software. There are two different groups of fault tolerance techniques: single version and multi-version software techniques [29]. The former includes program modularity, system closure, atomicity of actions, error detection, exception handling, checkpoint and restart, process pairs, and data diversity [44]; while the latter, so-called design diversity, is employed where multiple software versions are developed independently by different program teams using different design methods, yet they provide equivalent services according to the same requirement specifications. The main techniques of this multiple version software approach are recovery blocks, N-version programming, N self-checking programming, and other variants based on these three fundamental techniques.

Reliability models attempt to estimate the probability of coincident failures in multiple versions. Eckhardt and Lee (1985) [15] proposed the first reliability model of fault correlation in design diversity to observe positive correlations between version failures on the assumption of variation of difficulty on demand space. Littlewood and Miller (1989) [25] suggested that there was a possibility that negative fault correlations may exist on the basis of forced design diversity. Dugan and Lyu (1995) [14] proposed a Markov reward model to compare system reliability achieved by various design diversity approaches, and Tomek and Trivedi (1995) [43] suggested a Stochastic reward net model for software fault tolerance. Popov, Strigini et al. (2003) [37] estimated the upper and lower bounds for failure probability of design diversity based on the subdomain concept on the demand space. A detailed summary of fault-tolerant software and its reliability modeling methods can be found in [29]. Experimental comparisons and evaluations of some of the models are listed in [10] and [11].

3. Current trends and problems

The challenges in software reliability not only stem from the size, complexity, difficulty, and novelty of software applications in various domains, but also relate to the knowledge, training, experience and character of the software engineers involved. We address the current trends and problems from a number of software reliability engineering aspects.

3.1. Software reliability and system reliability

Although the nature of software faults is different from that of hardware faults, the theoretical foundation of software reliability comes from hardware reliability techniques. Previous work has been focused on extending the classical reliability theories from hardware to software, so that by employing familiar mathematical modeling schemes, we can establish software reliability framework consistently from the same viewpoints as hardware. The advantages of such modeling approaches are: (1) The physical meaning of the failure mechanism can be properly interpreted, so that the effect of failures on reliability, as measured in the form of failure rates, can be directly applied to the reliability models. (2) The combination of hardware reliability and software reliability to form system reliability models and measures can be provided in a unified theory. Even though the actual mechanisms of the various causes of hardware faults and software faults may be different, a single formulation can be employed from the reliability modeling and statistical estimation viewpoints. (3) System reliability models inherently engage system structure and modular design in block diagrams. The resulting reliability modeling process is not only intuitive (how components contribute to the overall reliability can be visualized), but also informative (reliability-critical components can be quickly identified).

The major drawbacks, however, are also obvious. First of all, while hardware failures may occur independently (or approximately so), software failures

do not happen independently. The interdependency of software failures is also very hard to describe in detail or to model precisely. Furthermore, similar hardware systems are developed from similar specifications, and hardware failures, usually caused by hardware defects, are repeatable and predictable. On the other hand, software systems are typically "one-of-a-kind." Even similar software systems or different versions of the same software can be based on quite different specifications. Consequently, software failures, usually caused by human design faults, seldom repeat in exactly the same way or in any predictable pattern. Therefore, while failure mode and effect analysis (FMEA) and failure mode and effect criticality analysis (FMECA) have long been established for hardware systems, they are not very well understood for software systems.

3.2. Software reliability modeling

Among all software reliability models, SRGM is probably one of the most successful techniques in the literature, with more than 100 models existing in one form or another, through hundreds of publications. In practice, however, SRGMs encounter major challenges. First of all, software testers seldom follow the operational profile to test the software, so what is observed during software testing may not be directly extensible for operational use. Secondly, when the number of failures collected in a project is limited, it is hard to make statistically meaningful reliability predictions. Thirdly, some of the assumptions of SRGM are not realistic, e.g., the assumptions that the faults are independent of each other; that each fault has the same chance to be detected in one class; and that correction of a fault never introduces new faults [40]. Nevertheless, the above setbacks can be overcome with suitable means. Given proper data collection processes to avoid drastic invalidation of the model assumptions, it is generally possible to obtain accurate estimates of reliability and to know that these estimates are accurate.

Although some historical SRGMs have been widely adopted to predict software reliability, researchers believe they can further improve the prediction accuracy of these models by adding other important factors which affect the final software quality [12,31,48]. Among others, code coverage is a metric commonly engaged by software testers, as it indicates how completely a test set executes a software system under test, therefore influencing the resulting reliability measure. To incorporate the effect of code coverage on reliability in the traditional software reliability models, [12] proposes a technique using both time and code coverage measurement for reliability prediction. It reduces the execution time by a parameterized factor when the test case neither increases code coverage nor causes a failure. These models, known as adjusted Non-Homogeneous Poisson Process (NHPP) models, have been shown empirically to achieve more accurate predictions than the original ones.

In the literature, several models have been proposed to determine the relationship between the number of failures/faults and the test coverage achieved, with various distributions. [48] suggests that this relation is a variant of the Rayleigh distribution, while [31] shows that it can be expressed as a logarithmic-exponential formula, based on the assumption that both fault coverage and test coverage follow the logarithmic NHPP growth model with respect to the execution time. More metrics can be incorporated to further explore this new modeling avenue.

Although there are a number of successful SRE models, they are typically measurement-based models which are employed in isolation at the later stage of the software development process. Early software reliability prediction models are often too insubstantial, seldom executable, insufficiently formal to be analyzable, and typically not linked to the target system. Their impact on the resulting reliability is therefore modest. There is currently a need for a creditable end-to-end software reliability model that can be directly linked to reliability prediction from the very beginning, so as to establish a systematic SRE procedure that can be certified, generalized and refined.

3.3. Metrics and measurements

Metrics and measurements have been an important part of the software development process, not only for software project budget planning but also for software quality assurance purposes. As software complexity and software quality are highly related to software reliability, the measurements of software complexity and quality attributes have been explored for early prediction of software reliability [39]. Static as well as dynamic program complexity measurements have been collected, such as lines of code, number of operators, relative program complexity, functional complexity, operational complexity, and so on. The complexity metrics can be further included in software reliability models for early reliability prediction, for example, to predict the initial software fault density and failure rate.

In SRGM, the two measurements related to reliability are: 1) the number of failures in a time period; and 2) time between failures. An important advancement of

SRGM is the notation of "time" during which failure data are recorded. It is demonstrated that CPU time is more suitable and more accurate than calendar time for recording failures, in which the actual execution time of software can be faithfully represented [35]. More recently, other forms of metrics for testing efforts have been incorporated into software reliability modeling to improve the prediction accuracy [8,18].

One key problem about software metrics and measurements is that they are not consistently defined and interpreted, again due to the lack of physical attributes of software. The achieved reliability measures may differ for different applications, yielding inconclusive results. A unified ontology to identify, describe, incorporate and understand reliability-related software metrics is therefore urgently needed.

3.4. Data collection and analysis

The software engineering process is described sardonically as a garbage-in/garbage-out process. That is to say, the accuracy of its output is bounded by the precision of its input. Data collection, consequently, plays a crucial role for the success of software reliability measurement.

There is an apparent trade-off between the data collection and the analysis effort. The more accuracy is required for analysis, the more effort is required for data collection. Fault-based data are usually easier to collect due to their static nature. Configuration management tools for source code maintenance can help to collect these data as developers are required to check in and check out new updated versions of code for fault removal. Failure-based data, on the other hand, are much harder to collect and usually require additional effort, for the following reasons. First, the dynamic operating condition where the failures occur may be hard to identify or describe. Moreover, the time when the failures occur must be recorded manually, after the failures are manifested. Calendar time data can be coarsely recorded, but they lack accuracy for modeling purposes. CPU time data, on the other hand, are very difficult to collect, particularly for distributed systems and networking environment where multiple CPUs are executing software in parallel. Certain forms of approximation are required to avoid the great pain in data collection, but then the accuracy of the data is consequently reduced. It is noted that while manual data collection can be very labor intensive, automatic data collection, although unavoidable, may be too intrusive (e.g., online collection of data can cause interruption to the system under test).

The amounts and types of data to be collected for reliability analysis purposes vary between organizations. Consequently, the experiences and lessons so gained may only be shared within the same company culture or at a high level of abstraction between organizations. To overcome this disadvantage, systematic failure data analysis for SRE purposes should be conducted.

Given field failure data collected from a real system, the analysis consists of five steps: 1) preprocessing of data, 2) analysis of data, 3) model structure identification and parameter estimation, 4) model solution, if necessary, and 5) analysis of models. In Step 1, the necessary information is extracted from the field data. The processing in this step requires detailed understanding of the target software and operational conditions. The actual processing required depends on the type of data. For example, the information in human-generated reports is usually not completely formatted. Therefore, this step involves understanding the situations described in the reports and organizing the relevant information into a problem database. In contrast, the information in automatically generated event logs is already formatted. Data processing of event logs consists of extracting error events and coalescing related error events.

In Step 2, the data are interpreted. Typically, this step begins with a list of measures to evaluate. However, new issues that have a major impact on software reliability can also be identified during this step. The results from Step 2 are reliability characteristics of operational software in actual environments and issues that must be addressed to improve software reliability. These include fault and error classification, error propagation, error and failure distribution, software failure dependency, hardware-related software errors, evaluation of software fault tolerance, error recurrence, and diagnosis of recurrences.

In Step 3, appropriate models (such as Markov models) are identified based on the findings from Step 2. We identify model structures and realistic ranges of parameters. The identified models are abstractions of the software reliability behavior in real environments. Statistical analysis packages and measurement-based reliability analysis tools are useful at this stage.

Step 4 involves either using known techniques or developing new ones to solve the model. Model solution allows us to obtain measures, such as reliability, availability, and performability. The results obtained from the model must be validated against real data. Reliability and performance modeling and

evaluation tools such as SHARPE [45] can be used in this step.

In Step 5, "what if" questions are addressed, using the identified models. Model factors are varied and the resulting effects on software reliability are evaluated. Reliability bottlenecks are determined and the effects of design changes on software reliability are predicted. Research work currently addressed in this area includes software reliability modeling in the operational phase, the modeling of the impact of software failures on performance, detailed error and recovery processes, and software error bursts. The knowledge and experience gained through such analysis can be used to plan additional studies and to develop the measurement techniques.

3.5. Methods and tools

In addition to software reliability growth modeling, many other methods are available for SRE. We provide a few examples of these methods and tools.

Fault trees provide a graphical and logical framework for a systematic analysis of system failure modes. Software reliability engineers can use them to assess the overall impact of software failures on a system, or to prove that certain failure modes will not occur. If they may occur, the occurrence probability can also be assessed. Fault tree models therefore provide an informative modeling framework that can be engaged to compare different design alternatives or system architectures with respect to reliability. In particular, they have been applied to both fault tolerant and fault intolerant (i.e., non-redundant) systems. Since this technique originates from hardware systems and has been extended to software systems, it can be employed to provide a unified modeling scheme for hardware/software co-design. Reliability modeling for hardware-software interactions is currently an area of intensive research [42].

In addition, simulation techniques can be provided for SRE purposes. They can produce observables of interest in reliability engineering, including discrete integer-valued quantities that occur as time progresses. One simulation approach produces artifacts in an actual software environment according to factors and influences believed to typify these entities within a given context [47]. The artifacts and environment are allowed to interact naturally, whereupon the flow of occurrences of activities and events is observed. This artifact-based simulation allows experiments to be set up to examine the nature of the relationships between software failures and other software metrics, such as program structure, programming error characteristics, and test strategies. It is suggested that the extent to which reliability depends merely on these factors can be measured by generating random programs having the given characteristics, and then observing their failure statistics.

Another reliability simulation approach [28] produces time-line imitations of reliability-related activities and events. Reliability measures of interest to the software process are modeled parametrically over time. The key to this approach is a rate-based architecture, in which phenomena occur naturally over time as controlled by their frequencies of occurrence, which depend on driving software metrics such as number of faults so far exposed or yet remaining, failure criticality, workforce level, test intensity, and software execution time. Rate-based event simulation is an example of a form of modeling called system dynamics, whose distinctive feature is that the observables are discrete events randomly occurring in time. Since many software reliability growth models are also based on rate (in terms of software hazard), the underlying processes assumed by these models are fundamentally the same as the rate-based reliability simulation. In general, simulations enable investigations of questions too difficult to be answered analytically, and are therefore more flexible and more powerful.

Various SRE measurement tools have been developed for data collection, reliability analysis, parameter estimation, model application and reliability simulation. Any major improvement on SRE is likely to focus on such tools. We need to provide tools and environments which can assist software developers to build reliable software for different applications. The partition of tools, environments, and techniques that will be engaged should reflect proper employment of the best current SRE practices.

3.6. Testing effectiveness and code coverage

As a typical mechanism for fault removal in software reliability engineering, software testing has been widely practiced in industry for quality assurance and reliability improvement. Effective testing is defined as uncovering of most if not all detectable faults. As the total number of inherent faults is not known, testing effectiveness is usually represented by a measurable testing index. Code coverage, as an indicator to show how thoroughly software has been stressed, has been proposed and is widely employed to represent fault coverage.

	Reference	Findings
Positive	Horgan (1994) [17] Frankl (1988) [16] Rapps (1988) [38]	High code coverage brings high software reliability and low fault rate.
	Chen (1992) [13]	A correlation between code coverage and software reliability was observed.
	Wong (1994)	The correlation between test effectiveness and block coverage is higher than that between test effectiveness and the size of test set.
	Frate (1995)	An increase in reliability comes with an increase in at least one code coverage measure, and a decrease in reliability is accompanied by a decrease in at least one code coverage measure.
	Cai (2005) [8]	Code coverage contributes to a noticeable amount of fault coverage.
Negative	Briand (2000) [6]	The testing result for published data did not support a causal dependency between code coverage and fault coverage.

Table 1. Comparison of Investigations on the Relation of Code Coverage to Fault Coverage

Despite the observations of a correlation between code coverage and fault coverage, a question is raised: Can this phenomenon of concurrent growth be attributed to a causal dependency between code coverage and fault detection, or is it just coincidental due to the cumulative nature of both measures? In one investigation of this question, an experiment involving Monte Carlo simulation was conducted on the assumption that there is no causal dependency between code coverage and fault detection [6]. The testing result for published data did not support a causal dependency between code coverage and defect coverage.

Nevertheless, many researchers consider coverage as a faithful indicator of the effectiveness of software testing results. A comparison among various studies on the impact of code coverage on software reliability is shown in Table 1.

3.7. Testing and operational profiles

The operational profile is a quantitative characterization of how a system will be used in the field by customers. It helps to schedule test activities, generate test cases, and select test runs. By allocating development and test resources to functions on the basis of how they are used, software reliability engineering can thus be planned with productivity and economics considerations in mind.

Using an operational profile to guide system testing ensures that if testing is terminated and the software is shipped because of imperative schedule constraints, the most-used operations will have received the most testing, and the reliability level will be the maximum that is practically achievable for the given test time. Also, in guiding regression testing, the profile tends to find, among the faults introduced by changes, the ones that have the most effect on reliability. Examples of the benefits of applying operational profiles can be found in a number of industrial projects [34].

Although significant improvement can be achieved by employing operational profiles in regression or system testing, challenges still exist for this technique. First of all, the operational profiles for some applications are hard to develop, especially for some distributed software systems, e.g., Web services. Moreover, unlike those of hardware, the operational profiles of software cannot be duplicated in order to speed the testing, because the failure behavior of software depends greatly on its input sequence and internal status. While in unit testing, different software units can be tested at the same time, this approach is therefore not applicable in system testing or regression testing. As a result, learning to deal with improper operational profiles and the dependences within the operational profile are the two major problems in operational profile techniques.

3.8. Industry practice and concerns

Although some success stories have been reported, there is a lack of wide industry adoption for software reliability engineering across various applications. Software practitioners often see reliability as a cost rather than a value, an investment rather than a return. Often the reliability attribute of a product takes less priority than its functionality or innovation. When product delivery schedule is tight, reliability is often the first element to be squeezed.

The main reason for the lack of industry enthusiasm in SRE is because its cost-effectiveness is not clear. Current SRE techniques incur visible overhead but yield invisible benefits. In contrast, a company's target is to have visible benefit but invisible overhead. The former requires some demonstration in the form of successful projects, while the latter involves avoidance

of labor-intensive tasks. Many companies, voluntarily or under compulsion from their quality control policy, collect failure data and make reliability measurements. They are not willing to spend much effort on data collection, let alone data sharing. Consequently, reliability results cannot be compared or benchmarked, and the experiences are hard to accumulate. Most software practitioners only employ some straightforward methods and metrics for their product reliability control. For example, they may use some general guidelines for quality metrics, such as fault density, lines of code, or development or testing time, and compare current projects with previous ones.

As the competitive advantage of product reliability is less obvious than that of other product quality attributes (such as performance or usability), few practitioners are willing to try out emerging techniques on SRE. The fact that there are so many software reliability models to choose from also intimidates practitioners. So instead of investigating which models are suitable for their environments or which model selection criteria can be applied, practitioners tend to simply take reliability measurements casually, and they are often suspicious about the reliability numbers obtained by the models. Many software projects claim to set reliability objectives such as five 9's or six 9's (meaning 0.99999 to 0.999999 availability or 10^{-5} to 10^{-6} failures per execution hour), but few can validate their reliability achievement.

Two major successful hardware reliability engineering techniques, reliability prediction by system architecture block diagrams and FME(C)A, still cannot be directly applied to software reliability engineering. This, as explained earlier, is due to the intricate software dependencies within and between software components (and sub-systems). If software components can be decoupled, or their dependencies can be clearly identified and properly modeled, then these popular techniques in hardware may be applicable to software, whereupon wide industry adoption may occur. We elaborate this in the following section.

3.9. Software architecture

Systematic examination of software architectures for a better way to support software development has been an active research direction in the past 10 years, and it will continue to be center stage in the coming decade [41]. Software architectural design not only impacts software development activities, but also affects SRE efforts. Software architecture should be enhanced to decrease the dependency of different software pieces that run on the same computer or platform so that their reliability does not interact. Fault isolation is a major design consideration for software architecture. Good software architecture should enjoy the property that exceptions are raised when faults occur, and module failures are properly confined without causing system failures. In particular, this type of component-based software development approach requires different framework, quality assurance paradigm [9], and reliability modeling [51] from those in traditional software development.

A recent trend in software architecture is that as information engineering is becoming the central focus for today's businesses, service-oriented systems and the associated software engineering will be the de facto standards for business development. Service orientation requires seamless integration of heterogeneous components and their interoperability for proper service creation and delivery. In a service-oriented framework, new paradigms for system organizations and software architectures are needed for ensuring adequate decoupling of components, swift discovery of applications, and reliable delivery of services. Such emerging software architectures include cross-platform techniques [5], open-world software [3], service-oriented architectures [32], and Web applications [22]. Although some modeling approaches have been proposed to estimate the reliability for specific Web systems [49], SRE techniques for general Web services and other service-oriented architectures require more research work.

4. Possible future directions

SRE activities span the whole software lifecycle. We discuss possible future directions with respect to five areas: software architecture, design, testing, metrics and emerging applications.

4.1. Reliability for software architectures and off-the-shelf components

Due to the ever-increasing complexity of software systems, modern software is seldom built from scratch. Instead, reusable components have been developed and employed, formally or informally. On the one hand, revolutionary and evolutionary object-oriented design and programming paradigms have vigorously pushed software reuse. On the other hand, reusable software libraries have been a deciding factor regarding whether a software development environment or methodology would be popular or not. In the light of this shift, reliability engineering for software development is

focusing on two major aspects: software architecture, and component-based software engineering.

The software architecture of a system consists of software components, their external properties, and their relationships with one another. As software architecture is the foundation of the final software product, the design and management of software architecture is becoming the dominant factor in software reliability engineering research. Well-designed software architecture not only provides a strong, reliable basis for the subsequent software development and maintenance phases, but also offers various options for fault avoidance and fault tolerance in achieving high reliability. Due to the cardinal importance of, and complexity involved in, software architecture design and modeling, being a good software architect is a rare talent that is highly demanded. A good software architect sees widely and thinks deeply, as the components should eventually fit together in the overall framework, and the anticipation of change has to be considered in the architecture design. A clean, carefully laid out architecture requires up-front investments in various design considerations, including high cohesion, low coupling, separation of modules, proper system closure, concise interfaces, avoidance of complexity, etc. These investments, however, are worthwhile since they eventually help to increase software reliability and reduce operation and maintenance costs.

One central research issue for software architecture concerning reliability is the design of failure-resilient architecture. This requires an effective software architecture design which can guarantee separation of components when software executes. When component failures occur in the system, they can then be quickly identified and properly contained. Various techniques can be explored in such a design. For example, memory protection prevents interference and failure propagation between different application processes. Guaranteed separation between applications has been a major requirement for the integration of multiple software services in complicated modern systems. It should be noted that the separation methods can support one another, and usually they are combined for achieve better reliability returns. Exploiting this synergy for reliability assessment is a possibility for further exploration.

In designing failure-resilient architecture, additional resources and techniques are often engaged. For example, error handling mechanisms for fault detection, diagnosis, isolation, and recovery procedures are incorporated to tolerate component failures; however, these mechanisms will themselves have some impact on the system. Software architecture has to take this impact into consideration. On the one hand, the added reliability-enhancement routines should not introduce unnecessary complexity, making them error-prone, which would decrease the reliability instead of increasing it. On the other hand, these routines should be made unintrusive while they monitor the system, and they should not further jeopardize the system while they are carrying out recovery functions. Designing concise, simple, yet effective mechanisms to perform fault detection and recovery within a general framework is an active research topic for researchers.

While software architecture represents the product view of software systems, component-based software engineering addresses the process view of software engineering. In this popular software development technique, many research issues are identified, such as the following. How can reliable general reusable components be identified and designed? How can existing components be modified for reusability? How can a clean interface design be provided for components so that their interactions are fully under control? How can defensive mechanisms be provided for the components so that they are protected from others, and will not cause major failures? How can it be determined whether a component is risk-free? How can the reliability of a component be assessed under untested yet foreseeable operational conditions? How can the interactions of components be modeled if they cannot be assumed independent? Component-based software engineering allows structure-based reliability to be realized, which facilitates design for reliability before the software is implemented and tested. The dependencies among components will thus need to be properly captured and modeled first.

These methods favor reliability engineering in multiple ways. First of all, they directly increase reliability by reducing the frequency and severity of failures. Run-time protections may also detect faults before they cause serious failures. After failures, they make fault diagnosis easier, and thus accelerate reliability improvements. For reliability assessment, these failure prevention methods reduce the uncertainties of application interdependencies or unexpected environments. So, for instance, having sufficient separation between running applications ensures that when we port an application to a new platform, we can trust its failure rate to equal that experienced in a similar use on a previous platform plus that of the new platform, rather than being also affected by the specific combination of other applications present on the new platform. Structure-

based reliability models can then be employed with this system aspect in place. With this modeling framework assisted by well-engineered software architecture, the range of applicability of structure-based models can further be increased. Examples of new applications could be to specify and investigate failure dependence between components, to cope with wide variations of reliability depending on the usage environment, and to assess the impact of system risk when components are checked-in or checked-out of the system.

4.2. Achieving design for reliability

To achieve reliable system design, fault tolerance mechanism needs to be in place. A typical response to system or software faults during operation includes a sequence of stages: Fault confinement, Fault detection, Diagnosis, Reconfiguration, Recovery, Restart, Repair, and Reintegration. Modern software systems pose challenging research issues in these stages, which are described as follows:

1. *Fault confinement.* This stage limits the spread of fault effects to one area of the system, thus preventing contamination of other areas. Fault-confinement can be achieved through use of self-checking acceptance tests, exception handling routines, consistency checking mechanisms, and multiple requests/confirmations. As the erroneous system behaviours due to software faults are typically unpredictable, reduction of dependencies is the key to successful confinement of software faults. This has been an open problem for software reliability engineering, and will remain a tough research challenge.

2. *Fault detection.* This stage recognizes that something unexpected has occurred in the system. Fault latency is the period of time between the occurrence of a software fault and its detection. The shorter it is, the better the system can recover. Techniques fall in two classes: off-line and on-line. Off-line techniques such as diagnostic programs can offer comprehensive fault detection, but the system cannot perform useful work while under test. On-line techniques, such as watchdog monitors or redundancy schemes, provide a real-time detection capability that is performed concurrently with useful work.

3. *Diagnosis.* This stage is necessary if the fault detection technique does not provide information about the failure location and/or properties. On-line, failure-prevention diagnosis is the research trend. When the diagnosis indicates unhealthy conditions in the system (such as low available system resources), software rejuvenation can be performed to achieve in-time transient failure prevention.

4. *Reconfiguration.* This stage occurs when a fault is detected and a permanent failure is located. The system may reconfigure its components either to replace the failed component or to isolate it from the rest of the system. Successful reconfiguration requires robust and flexible software architecture and the associated reconfiguration schemes.

5. *Recovery.* This stage utilizes techniques to eliminate the effects of faults. Two basic recovery approaches are based on: fault masking, retry and rollback. Fault-masking techniques hide the effects of failures by allowing redundant, correct information to outweigh the incorrect information. To handle design (permanent) faults, N-version programming can be employed. Retry, on the other hand, attempts a second try at an operation and is based on the premise that many faults are transient in nature. A recovery blocks approach is engaged to recover from software design faults in this case. Rollback makes use of the system operation having been backed up (checkpointed) to some point in its processing prior to fault detection and operation recommences from this point. Fault latency is important here because the rollback must go back far enough to avoid the effects of undetected errors that occurred before the detected error. The effectiveness of design diversity as represented by N-version programming and recovery blocks, however, continues to be actively debated.

6. *Restart.* This stage occurs after the recovery of undamaged information. Depending on the way the system is configured, hot restart, warm restart, or cold restart can be achieved. In hot restart, resumption of all operations from the point of fault detection can be attempted, and this is possible only if no damage has occurred. In warm restart, only some of the processes can be resumed without loss; while in cold restart, complete reload of the system is performed with no processes surviving.

7. *Repair.* In this stage, a failed component is replaced. Repair can be off-line or on-line. In off-line repair, if proper component isolation can be achieved, the system will continue as the failed component can be removed for operation. Otherwise, the system must be brought down to perform the repair, and so the system availability and reliability depends on how fast a fault can be located and removed. In on-line repair the component may be replaced immediately with a backup spare (in a procedure equivalent to reconfiguration) or operation may continue without the faulty component (for example, masking redundancy

or graceful degradation). With on-line repair, system operation is not interrupted; however, achieving complete and seamless repair poses a major challenge to researchers.

8. *Reintegration.* In this stage the repaired module must be reintegrated into the system. For on-line repair, reintegration must be performed without interrupting system operation.

Design for reliability techniques can further be pursued in four different areas: fault avoidance, fault detection, masking redundancy, and dynamic redundancy. Non-redundant systems are fault intolerant and, to achieve reliability, generally use fault avoidance techniques. Redundant systems typically use fault detection, masking redundancy, and dynamic redundancy to automate one or more of the stages of fault handling. The main design consideration for software fault tolerance is cost-effectiveness. The resulting design has to be effective in providing better reliability, yet it should not introduce excessive cost, including performance penalty and unwarranted complexity, which may eventually prove unworthy of the investigation.

4.3. Testing for reliability assessment

Software testing and software reliability have traditionally belonged to two separate communities. Software testers test software without referring to how software will operate in the field, as often the environment cannot be fully represented in the laboratory. Consequently they design test cases for exceptional and boundary conditions, and they spend more time trying to break the software than conducting normal operations. Software reliability measurers, on the other hand, insist that software should be tested according to its operational profile in order to allow accurate reliability estimation and prediction. In the future, it will be important to bring the two groups together, so that on the one hand, software testing can be effectively conducted, while on the other hand, software reliability can be accurately measured. One approach is to measure the test compression factor, which is defined as the ratio between the mean time between failures during operation and during testing. This factor can be empirically determined so that software reliability in the field can be predicted from that estimated during testing. Another approach is to ascertain how other testing related factors can be incorporated into software reliability modeling, so that accurate measures can be obtained based on the effectiveness of testing efforts.

Recent studies have investigated the effect of code coverage on fault detection under different testing profiles, using different coverage metrics, and have studied its application in reducing test set size [30]. Experimental data are required to evaluate code coverage and determine whether it is a trustworthy indicator for the effectiveness of a test set with respect to fault detection capability. Also, the effect of code coverage on fault detection may vary under different testing profiles. The correlation between code coverage and fault coverage should be examined across different testing schemes, including function testing, random testing, normal testing, and exception testing. In other words, white box testing and black box testing should be cross–checked for their effectiveness in exploring faults, and thus yielding reliability increase.

Furthermore, evidence for variation between different coverage metrics can also established. Some metrics may be independent and some correlated. The quantitative relationship between different code coverage metrics and fault detection capability should be assessed, so that redundant metrics can be removed, and orthogonal ones can be combined. New findings about the effect of code coverage and other metrics on fault detection can be used to guide the selection and evaluation of test cases under various testing profiles, and a systematic testing scheme with predictable reliability achievement can therefore be derived.

Reducing test set size is a key goal in software testing. Different testing metrics should be evaluated regarding whether they are good filters in reducing the test set size, while maintaining the same effectiveness in achieving reliability. This assessment should be conducted under various testing scenarios [8]. If such a filtering capability can be established, then the effectiveness of test cases can be quantitatively determined when they are designed. This would allow the prediction of reliability growth with the creation a test set before it is executed on the software, thus facilitating early reliability prediction and possible feedback control for better test set design schemes.

Other than linking software testing and reliability with code coverage, statistical learning techniques may offer another promising avenue to explore. In particular, statistical debugging approaches [26, 52], whose original purpose was to identify software faults with probabilistic modeling of program predicates, can provide a fine quantitative assessment of program codes with respect to software faults. They can therefore help to establish accurate software reliability

prediction models based on program structures under testing.

4.4. Metrics for reliability prediction

Today it is almost a mandate for companies to collect software metrics as an indication of a maturing software development process. While it is not hard to collect metrics data, it is not easy to collect clean and consistent data. It is even more difficult to derive meaningful results from the collected metrics data. Collecting metrics data for software reliability prediction purposes across various projects and applications is a major challenge. Moreover, industrial software engineering data, particularly those related to system failures, are historically hard to obtain across a range of organizations. It will be important for a variety of sources (such as NASA, Microsoft, IBM, Cisco, etc.) across industry and academia to make available real-failure data for joint investigation to establish credible reliability analysis procedures. Such a joint effort should define (1) what data to collect by considering domain sensitivities, accessibility, privacy, and utility; (2) how to collect data in terms of tools and techniques; and (3) how to interpret and analyze the data using existing techniques.

In addition to industrial data collection efforts, novel methods to improve reliability prediction are actively being researched. For example, by extracting rich information from metrics data using a sound statistical and probability foundation, Bayesian Belief Networks (BBNs) offer a promising direction for investigation in software engineering [7]. BBNs provide an attractive formalism for different software cases. The technique allows software engineers to describe prior knowledge about software development quality and software verification and validation (SV&V) quality, with manageable visual descriptions and automated inferences. The software reliability process can then be modified with inference from observed failures, and future reliability can be predicted. With proper engagement of software metrics, this is likely to be a powerful tool for reliability assessment of software based systems, finding applications in predicting software defects, forecasting software reliability, and determining runaway projects [1].

Furthermore, traditional reliability models can be enhanced to incorporate some testing completeness or effectiveness metrics, such as code coverage, as well as their traditional testing-time based metrics. The key idea is that failure detection is not only related to the time that the software is under testing, but also what fraction of the code has been executed by the testing. The effect of testing time on reliability can be estimated using distributions from traditional SRGMs. However, new models are needed to describe the effect of coverage on reliability. These two dimensions, testing time and coverage, are not orthogonal. The degree of dependency between them is thus an open problem for investigation. Formulation of new reliability models which integrate time and coverage measurements for reliability prediction would be a promising direction.

One drawback of the current metrics and data collection process is that it is a one-way, open-loop avenue: while metrics of the development process can indicate or predict the outcome quality, such as the reliability, of the resulting product, they often cannot provide feedback to the process regarding how to make improvement. Metrics would present tremendous benefits to reliability engineering if they could achieve not just prediction, but also refinement. Traditional software reliability models take metrics (such as defect density or times between failures) as input and produce reliability quantity as the output. In the future, a reverse function is urgently called for: given a reliability goal, what should the reliability process (and the resulting metrics) look like? By providing such feedback, it is expected that a closed-loop software reliability engineering process can be informative as well as beneficial in achieving predictably reliable software.

4.5. Reliability for emerging software applications

Software engineering targeted for general systems may be too ambitious. It may find more successful applications if it is domain-specific. In this Future of Software Engineering volume, future software engineering techniques for a number of emerging application domains have been thoroughly discussed. Emerging software applications also create abundant opportunities for domain-specific reliability engineering.

One key industry in which software will have a tremendous presence is the service industry. Service-oriented design has been employed since the 1990s in the telecommunications industry, and it reached software engineering community as a powerful paradigm for Web service development, in which standardized interfaces and protocols gradually enabled the use of third-party functionality over the Internet, creating seamless vertical integration and enterprise process management for cross-platform, cross-provider, and cross-domain applications. Based

on the future trends for Web application development as laid out in [22], software reliability engineering for this emerging technique poses enormous challenges and opportunities. The design of reliable Web services and the assessment of Web service reliability are novel and open research questions. On the one hand, having abundant service providers in a Web service makes the design diversity approach suddenly appealing, as the diversified service design is perceived not as cost, but as an available resource. On the other hand, this unplanned diversity may not be equipped with the necessary quality, and the compatibility among various service providers can pose major problems. Seamless Web service composition in this emerging application domain is therefore a central issue for reliability engineering. Extensive experiments are required in the area of measurement of Web service reliability. Some investigations have been initiated with limited success [27], but more efforts are needed.

Researchers have proposed the publish/subscribe paradigm as a basis for middleware platforms that support software applications composed of highly evolvable and dynamic federations of components. In this approach, components do not interact with each other directly; instead an additional middleware mediates their communications. Publish/subscribe middleware decouples the communication among components and supports implicit bindings among components. The sender does not know the identity of the receivers of its messages, but the middleware identifies them dynamically. Consequently new components can dynamically join the federation, become immediately active, and cooperate with the other components without requiring any reconfiguration of the architecture. Interested readers can refer to [21] for future trends in middleware-based software engineering technologies.

The open system approach is another trend in software applications. Closed-world assumptions do not hold in an increasing number of cases, especially in ubiquitous and pervasive computing settings, where the world is intrinsically open. Applications cover a wide range of areas, from dynamic supply-chain management, dynamic enterprise federations, and virtual endeavors, on the enterprise level, to automotive applications and home automation on the embedded-systems level. In an open world, the environment changes continuously. Software must adapt and react dynamically to changes, even if they are unanticipated. Moreover, the world is open to new components that context changes could make dynamically available – for example, due to mobility. Systems can discover and bind such components dynamically to the application while it is executing. The software must therefore exhibit a self-organization capability. In other words, the traditional solution that software designers adopted – carefully elicit change requests, prioritize them, specify them, design changes, implement and test, then redeploy the software – is no longer viable. More flexible and dynamically adjustable reliability engineering paradigms for rapid responses to software evolution are required.

5. Conclusions

As the cost of software application failures grows and as these failures increasingly impact business performance, software reliability will become progressively more important. Employing effective software reliability engineering techniques to improve product and process reliability would be the industry's best interests as well as major challenges. In this paper, we have reviewed the history of software reliability engineering, the current trends and existing problems, and specific difficulties. Possible future directions and promising research problems in software reliability engineering have also been addressed. We have laid out the current and possible future trends for software reliability engineering in terms of meeting industry and customer needs. In particular, we have identified new software reliability engineering paradigms by taking software architectures, testing techniques, and software failure manifestation mechanisms into consideration. Some thoughts on emerging software applications have also been provided.

References

[1] S. Amasaki, O. Mizuno, T. Kikuno, and Y. Takagi, "A Bayesian Belief Network for Predicting Residual Faults in Software Products," *Proceedings of 14th International Symposium on Software Reliability Engineering (ISSRE2003),* November 2003, pp. 215-226,

[2] ANSI/IEEE, Standard Glossary of Software Engineering Terminology, *STD-729-1991, ANSI/IEEE*, 1991.

[3] L. Baresi, E. Nitto, and C. Ghezzi, "Toward Open-World Software: Issues and Challenges," *IEEE Computer*, October 2006, pp. 36-43.

[4] A. Bertolino, "Software Testing Research: Achievements, Challenges, Dreams," *Future of Software Engineering 2007*, L. Briand and A. Wolf (eds.), IEEE-CS Press, 2007.

[5] J. Bishop and N. Horspool, "Cross-Platform Development: Software That Lasts," *IEEE Computer*, October 2006, pp. 26-35.

[6] L. Briand and D. Pfahl, "Using Simulation for Assessing the Real Impact of Test Coverage on Defect Coverage,"

IEEE Transactions on Reliability, vol. 49, no. 1, March 2000, pp. 60-70.

[7] J. Cheng, D.A. Bell, and W. Liu, "Learning Belief Networks from Data: An Information Theory Based Approach," *Proceedings of the Sixth International Conference on Information and Knowledge Management*, Las Vegas, 1997, pp. 325-331.

[8] X. Cai and M.R. Lyu, "The Effect of Code Coverage on Fault Detection Under Different Testing Profiles," *ICSE 2005 Workshop on Advances in Model-Based Software Testing (A-MOST)*, St. Louis, Missouri, May 2005.

[9] X. Cai, M.R. Lyu, and K.F. Wong, "A Generic Environment for COTS Testing and Quality Prediction," *Testing Commercial-off-the-shelf Components and Systems*, S. Beydeda and V. Gruhn (eds.), Springer-Verlag, Berlin, 2005, pp. 315-347.

[10] X. Cai, M.R. Lyu, and M.A. Vouk, "An Experimental Evaluation on Reliability Features of N-Version Programming," in *Proceedings 16th International Symposium on Software Reliability Engineering (ISSRE'2005)*, Chicago, Illinois, Nov. 8-11, 2005.

[11] X. Cai and M.R. Lyu, "An Empirical Study on Reliability and Fault Correlation Models for Diverse Software Systems," in *Proceedings 15th International Symposium on Software Reliability Engineering (ISSRE'2004)*, Saint-Malo, France, Nov. 2004, pp.125-136.

[12] M. Chen, M.R. Lyu, and E. Wong, "Effect of Code Coverage on Software Reliability Measurement," *IEEE Transactions on Reliability*, vol. 50, no. 2, June 2001, pp.165-170.

[13] M.H. Chen, A.P. Mathur, and V.J. Rego, "Effect of Testing Techniques on Software Reliability Estimates Obtained Using Time Domain Models," In *Proceedings of the 10th Annual Software Reliability Symposium*, Denver, Colorado, June 1992, pp. 116-123.

[14] J.B. Dugan and M.R. Lyu, "Dependability Modeling for Fault-Tolerant Software and Systems," in *Software Fault Tolerance*, M. R. Lyu (ed.), New York: Wiley, 1995, pp. 109–138.

[15] D.E. Eckhardt and L.D. Lee, "A Theoretical Basis for the Analysis of Multiversion Software Subject to Coincident Errors," *IEEE Transactions on Software Engineering*, vol. 11, no. 12, December 1985, pp. 1511–1517.

[16] P.G. Frankl and E.J. Weyuker, "An Applicable Family of Data Flow Testing Criteria," *IEEE Transactions on Software Engineering*, vol. 14, no. 10, October 1988, pp. 1483-1498.

[17] J.R. Horgan, S. London, and M.R. Lyu, "Achieving Software Quality with Testing Coverage Measures," *IEEE Computer*, vol. 27, no.9, September 1994, pp. 60-69.

[18] C.Y. Huang and M.R. Lyu, "Optimal Release Time for Software Systems Considering Cost, Testing-Effort, and Test Efficiency," *IEEE Transactions on Reliability*, vol. 54, no. 4, December 2005, pp. 583-591.

[19] C.Y. Huang, M.R. Lyu, and S.Y. Kuo, "A Unified Scheme of Some Non-Homogeneous Poisson Process Models for Software Reliability Estimation," *IEEE Transactions on Software Engineering*, vol. 29, no. 3, March 2003, pp. 261-269.

[20] W.S. Humphrey, "The Future of Software Engineering: I," Watts New Column, News at SEI, vol. 4, no. 1, March, 2001.

[21] V. Issarny, M. Caporuscio, and N. Georgantas: "A Perspective on the Future of Middleware-Based Software Engineering," *Future of Software Engineering 2007*, L. Briand and A. Wolf (eds.), IEEE-CS Press, 2007.

[22] M. Jazayeri, "Web Application Development: The Coming Trends," *Future of Software Engineering 2007*, L. Briand and A. Wolf (eds.), IEEE-CS Press, 2007.

[23] Z. Jelinski and P.B. Moranda, "Software Reliability Research," in *Proceedings of the Statistical Methods for the Evaluation of Computer System Performance*, Academic Press, 1972, pp. 465-484.

[24] B. Littlewood and L. Strigini, "Software Reliability and Dependability: A Roadmap," in *Proceedings of the 22nd International Conference on Software Engineering (ICSE'2000),* Limerick, June 2000, pp. 177-188.

[25] B. Littlewood and D. Miller, "Conceptual Modeling of Coincident Failures in Multiversion Software," *IEEE Transactions on Software Engineering*, vol. 15, no. 12, December 1989, pp. 1596–1614.

[26] C. Liu, L. Fei, X. Yan, J. Han, and S. Midkiff, "Statistical Debugging: A Hypothesis Testing-based Approach," *IEEE Transaction on Software Engineering*, vol. 32, no. 10, October, 2006, pp. 831-848.

[27] N. Looker and J. Xu, "Assessing the Dependability of SOAP-RPC-Based Web Services by Fault Injection," in *Proceedings of 9th IEEE International Workshop on Object-oriented Real-time Dependable Systems*, 2003, pp. 163-170.

[28] M.R. Lyu (ed.), *Handbook of Software Reliability Engineering*, IEEE Computer Society Press and McGraw-Hill, 1996.

[29] M.R. Lyu and X. Cai, "Fault-Tolerant Software," *Encyclopedia on Computer Science and Engineering*, Benjamin Wah (ed.), Wiley, 2007.

[30] M.R. Lyu, Z. Huang, S. Sze, and X. Cai, "An Empirical Study on Testing and Fault Tolerance for Software Reliability Engineering," in *Proceedings 14th IEEE International Symposium on Software Reliability Engineering (ISSRE'2003)*, Denver, Colorado, November 2003, pp.119-130.

[31] Y.K. Malaiya, N. Li, J.M. Bieman, and R. Karcich, "Software Reliability Growth with Test Coverage," *IEEE*

Transactions on Reliability, vol. 51, no. 4, December 2002, pp. 420-426.

[32] T. Margaria and B. Steffen, "Service Engineering: Linking Business and IT," *IEEE Computer*, October 2006, pp. 45-55.

[33] J.D. Musa, *Software Reliability Engineering: More Reliable Software Faster and Cheaper (2nd Edition)*, AuthorHouse, 2004.

[34] J.D. Musa, "Operational Profiles in Software Reliability Engineering," *IEEE Software*, Volume 10, Issue 2, March 1993, pp. 14-32.

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

[36] H. Pham, *Software Reliability*, Springer, Singapore, 2000.

[37] P.T. Popov, L. Strigini, J. May, and S. Kuball, "Estimating Bounds on the Reliability of Diverse Systems," *IEEE Transactions on Software Engineering*, vol. 29, no. 4, April 2003, pp. 345–359.

[38] S. Rapps and E.J. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Transactions on Software Engineering*, vol. 11, no. 4, April 1985, pp. 367-375.

[39] Rome Laboratory (RL), *Methodology for Software Reliability Prediction and Assessment*, Technical Report RL-TR-92-52, volumes 1 and 2, 1992.

[40] M.L. Shooman, *Reliability of Computer Systems and Networks: Fault Tolerance, Analysis and Design*, Wiley, New York, 2002.

[41] R. Taylor and A. van der Hoek, "Software Design and Architecture: The Once and Future Focus of Software Engineering," *Future of Software Engineering 2007*, L. Briand and A. Wolf (eds.), IEEE-CS Press, 2007.

[42] X. Teng, H. Pham, and D. Jeske, "Reliability Modeling of Hardware and Software Interactions, and Its Applications," *IEEE Transactions on Reliability*, vol. 55, no. 4, Dec. 2006, pp. 571-577.

[43] L.A. Tomek and K.S. Trivedi, "Analyses Using Stochastic Reward Nets," in *Software Fault Tolerance*, M.R. Lyu (ed.), New York: Wiley, 1995, pp. 139–165.

[44] W. Torres-Pomales, "Software Fault Tolerance: A Tutorial," NASA Langley Research Center, Hampton, Virginia, TM-2000-210616, Oct. 2000.

[45] K.S. Trivedi, "SHARPE 2002: Symbolic Hierarchical Automated Reliability and Performance Evaluator," in *Proceedings International Conference on Dependable Systems and Networks*, 2002.

[46] K.S. Trivedi, K. Vaidyanathan, and K. Goseva-Postojanova, "Modeling and Analysis of Software Aging and Rejuvenation", in *Proceedings of 33 rd Annual Simulation Symposium*, IEEE Computer Society Press, Los Alamitos, CA, 2000, pp. 270-279.

[47] A. von Mayrhauser and D. Chen, "Effect of Fault Distribution and Execution Patterns on Fault Exposure in Software: A Simulation Study," *Software Testing, Verification & Reliability*, vol. 10, no.1, March 2000, pp. 47-64.

[48] M.A. Vouk, "Using Reliability Models During Testing With Nonoperational Profiles," in *Proceedings of 2nd Bellcore/Purdue Workshop on Issues in Software Reliability Estimation*, October 1992, pp. 103-111.

[49] W. Wang and M. Tang, "User-Oriented Reliability Modeling for a Web System," in *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE'03)*, Denver, Colorado, November 2003, pp.1-12.

[50] M. Xie, *Software Reliability Modeling*, World Scientific Publishing Company, 1991.

[51] S. Yacoub, B. Cukic, and H Ammar, "A Scenario-Based Reliability Analysis Approach for Component-Based Software," *IEEE Transactions on Reliability*, vol. 53, no. 4, 2004, pp. 465-480.

[52] A.X. Zheng, M.I. Jordan, B. Libit, M. Naik, and A. Aiken, "Statistical Debugging: Simultaneous Identification of Multiple Bugs," in *Proceedings of the 23rd International Conference on Machine Learning*, Pittsburgh, PA, 2006, pp. 1105-1112.