CHAPTER 1

INTRODUCTION

This chapter provides an introduction to the context of the research themes explored in the thesis. The chapter introduces the reader to the issues, challenges and existing techniques in Software Reliability through Models. It gives a high-level overview of various approaches to improve quality through Reliability Model, Intelligent Code Evaluator, and Reliability Factor Analysis and discusses various Enhancement Techniques. The chapter ends with an outline and contributions of the research work.

1.1. SOFTWARE QUALITY

Software Engineering Musa (1975) is the application of systematic, disciplined and quantifiable approach to the development, operation, and maintenance of Software. It is a direct subfield of Computer Science and has some relations with Management Science. Software development life cycle involves requirement analysis, design, coding, testing, and implementation. Software Quality is the most important one, since the success of a Software Engineer relies on the development of failure free software S. Inoue and Yamada (2003). Quality is defined as the bundle of attributes present in a commodity and, where appropriate, the level of the attribute for which the consumer (software users) holds a positive value. Defining the attributes of software quality and determining the metrics to assess the relative value of each attribute are not formalized processes. Compounding the problem is that
numerous metrics exist to test each quality attribute. As users place different values on each attribute depending on the product’s use, it is important that quality attributes be observable to consumers.

However, with software there exist not only asymmetric information problems (where a developer has more information about quality than the consumer), but also instances where the developer truly does not know the quality of his own product Chavez (2000). It is not unusual for software to become technically obsolete before its performance attributes have been fully demonstrated under real-world operating conditions. As software has evolved over time so has the definition of software quality attributes. Goel and Okumoto. (1979) first attempted to assess quality attributes for software. His software quality model characterizes Xie (2000) attributes in terms of three categories:

- product operation,
- product revision,
- and product transition.

In 1991, the International Organization for Standardization (ISO) adopted ISO 9126 as the standard for software quality (ISO 1991). It is structured around six main attributes listed below Tausworthe and Lyu (1996) (sub-characteristics are listed in parenthesis):

1. Functionality (suitability, accurateness, interoperability, compliance, security)
2. Reliability (maturity, fault tolerance, recoverability)
3. Usability (understandability, learnability, operability)

4. Efficiency (time behaviour, resource behaviour)

5. Maintainability (analyzability, changeability, stability, testability)

6. Portability (adaptability, installability, conformance, replaceability)

Although a general set of standards have been agreed on, the appropriate metrics to test how well software meets those standards are still poorly defined. Publications by IEEE (1988, 1996) have presented numerous potential metrics that can be used to test each attribute. These metrics include

1. Fault density,
2. Requirements compliance,
3. Test coverage, and
4. Mean time to failure.

The problem is that no one metric is able to unambiguously measure a particular quality attribute. Different metrics may give different rank orderings of the same attribute, making comparisons across products that are difficult and uncertain.

1.2 SOFTWARE RELIABILITY IN QUALITY IMPROVEMENT

Quality of the software is acted by external and internal variables. Operational mode of the software by user experience is through external quality Liao et al. (2004). Internal quality is dependent on coding. It is purely
based on coding aspects. Figure 1.1 shows the software quality improvement through these four processes.

**Figure 1.1 Software Quality Improvement**

Software and Hardware Reliability is based on modes of failure Lyu (1993). Hardware modes of failure wear, design flaws and uncertainty environmental phenomena are more tangible because hardware is a physical entity Ohba (1984). But Software does have a mode of failure, which is based on the assumption that development is not a perfect process. Failure occurs when the software does not perform according to the specification. It is important to recognize that there is a different between hardware failure rate and software failure rate. Hardware fault Tian and Noore (2005) is high in initial manufacturing but then decreases as the faulty components are identified and removed or if the components are stabilized. As the component physically wears out, the fault rate starts to increase. Software however, has a different fault or error identification rate.
For software, the error rate is at the highest level at integration and test. As it is tested, errors are identified and removed. This removal continues at a slower rate during its operational use; the number of errors continually decreasing, assuming no new errors are introduced Hu et al (2007) Software does not have moving parts and does not physically wear out as hardware, but it does outlive its usefulness and becomes obsolete. To quantify software reliability in a meaningful way, the use of software must be modelled as a random process in which a use is selected according to some probability distribution Jaynes (1963), or use distribution. Reliability then becomes the probability that the software will perform according to the given specification for a randomly selected use. When the software fails to meet the specification during use, a failure occurs.

So reliability can be a useful metric. This is useful for software development. Reliability can be defined in two different ways. Reliability as a function of time Huang and Lin (2006), perhaps the more traditional definition, addresses the design of software that will operate according to specification for a period of time. But a simpler definition can also be used – Reliability is the probability that a randomly chosen use (test case) will be processed correctly Antoniol et al (2004). Using this latter definition the mean time to failure is the average number of uses between failures. MTTF and reliability can be related mathematically Tseng (2005) in the models.

1.3 SOFTWARE TESTING

Software testing is the process of applying metrics to determine product quality. Software testing Okamura et al (2006) is the dynamic execution of software and the comparison of the results of that execution against a set of pre-determined criteria. “Execution” is the process of running
the software on a computer with or without any form of instrumentation or test control software being present. “Pre-determined criteria” means that the software’s capabilities are known prior to its execution. What the software actually does can then be compared against the anticipated results to judge whether the software behaved correctly Voas (2003)

In many respects, software testing is an infrastructure technology or “infratechnology.” Infra-technologies are technical tools, including scientific and engineering data, measurement and test methods, and practices and techniques that are widely used in industry Glass (1998). Software testing infra-technologies provide the tools needed to measure conformance, performance, and interoperability during the software development Littlewood and Strigini (2000). These tools aid in testing the relative performance of different software configurations and mitigate the expense of reengineering software after it is developed and released. Software testing infra-technologies also provide critical information to the software user regarding the quality of the software Goseva-Popstojanova and Trivedi (2000). By increasing the quality, purchase decision costs for software are reduced.

Software reliability Chen et al (2001) efforts and software testing process complements each other: The results of software testing provide statistical data to model the reliability, and the reliability level of the software determines the amount of necessary testing Ehrlich et al (1993). In order to provide reliability assessment process with healthy input data, the testing of software must be comprehensive and complete both in terms of user requirements and software architecture Musa (1975). While well-known software engineering sources Chen et al (2001) suggest ways to improve testing process, reliability oriented studies Goel and Okumoto (1979) are still
worth mentioning. The major difference between the viewpoints of “software engineers” and of “software reliability engineers” is that the former is mostly interested in the coverage of functionalities and flow paths, whereas the latter is interested in coverage of failures (or defects) Malaiya et al (2002).

There, however, exist some problems with software testing process when software reliability is of primary concern. The first problem with software testing is coverage: Due to the direct effect of the selection of failure data on the reliability model performance, the content and coverage of the tests are critical Lanubile (1996). Coverage problem also affects the cost of a project, since the cost of finding a defect in early phases of software development process is lower than that of finding it later in the development process Boland and Singh (2003). Another important aspect of test coverage is that the selection of test cases and failure data influences the way the software reliability estimation model are formed Lanubile (1996).

The second problem is detection and prevention of failures Karunanithi et al (1992); not every failure is an independent one and it is possible that removal of a failure also remove (or introduce) another one. That is why the nature of the failures should be investigated to see if there is correlation between failures Goševa-Popstojanova et al Trivedi (2000). At this point, the study of Wohlin and Korner gains importance. In that study a model has been formed to represent the spread of defects based on a level-approach, in which the term “level” corresponds to the phase of the development process that a specific fault is first introduced.

It is stated in that study that a defect found in a level can be the indicator of the defects in previous levels. In contrast to coverage of functionality, which is some sort of validation of what is intended to
implement, the business of failure coverage Souza and Vergilio (2006) is not a straight-forward action due to stochastic nature of distribution of failures. Wohlin and Korner’s method solves this problem up to some extent (1990). However their assumption that a failure in a level is independent of the others cause problem in real-life Goševa-Popstojanova and Trivedi (2000). In deed, the relation of a defect found in the early phases of the project with another one found in the later steps is not covered in their study.

An idea to relax the testing process, which is proposed by Boland and Singh (2003) is that the effect of finding an error in the early phases has more noticeable effect on the overall failure rate of the software than that of finding it later. That idea leads to the corollary that it is helpful to spend more effort on testing at early stages, beginning at component testing and code-review Wohlin and Korner (1990). There are some studies to determine a method to guarantee failure coverage. Some researches prefer use of test-coverage methods to defect-coverage and generate the concept of test-coverage growth. It is proven in another study that the ability to detect defects is correlated with code coverage. A method is formed in that study for this purpose and the results are compared with well-known software reliability growth models Karunanithi and Malaiya (1992) to determine their accuracy.

1.3.1 System Testing

System Testing is the process of performing a variety of tests on a system to explore functionality or to identify problems. System testing is usually required before and after a system is put in place. A series of systematic procedures are referred to while testing is being performed Malaiya et al (2002). These procedures tell the tester how the system should perform and where common mistakes may be found. Testers usually try to
“break the system” by entering data that may cause the system to malfunction or return incorrect information.

System testing of software or hardware is testing conducted on a complete, integrated system to evaluate the system’s compliance with its specified requirements Tian and Noore (2005). System testing falls within the scope of black box testing, and as such, should require no knowledge of the inner design of the code or logic. System testing is performed on the entire system in the context of a Functional Requirement Specification (FRS) or a System Requirement Specification (SRS) Vouk (1992). System testing tests not only the design, but also the behavior and even the believed expectations of the customer. It is also intended to test up to and beyond the bounds defined in the software & hardware requirements specification.

System testing is black box testing, performed by the test team, and at the start of the system testing the complete system is configured in a controlled environment. The purpose of system testing is to validate an application’s accuracy and completeness in performing the functions as designed. System testing simulates real life scenarios that occur in a “simulated real life” test environment, and tests all functions of the system that are required in real life. System testing is deemed complete when actual results and expected results are either in line or differences are explainable or acceptable, based on client input Tausworthe and Lyu (1996).

Several modules constitute a project. If the project is long-term project, several developers write the modules. Once all the modules are integrated, several errors may arise. The testing done at this stage is called system test.
System testing ensures that the entire integrated software system meets requirements. It tests a configuration to ensure known and predictable results. System testing is based on process descriptions and flows, emphasizing pre-driven process links and integration points.

1.3.2 Unit Testing

In computer programming, unit testing is a method by which individual units of source code are tested to determine if they are fit for use. A unit is the smallest testable part of an application. In procedural programming a unit may be an individual function or procedure. Unit tests are created by programmers or occasionally by white box testers.

Unit testing is a software development process in which the smallest testable parts of an application called units are individually and independently scrutinized for proper operation. Unit testing is often automated but it can also be done manually.

1.3.3 Functional Testing

Functional testing is a type of black box testing that bases its test cases on the specifications of the software component under test Rome Laboratory (RL) (1992). Functions are tested by feeding them input and examining the output, and internal program structure is rarely considered.

Functional testing differs from system testing in that functional testing Liu (2006) “verifies a program by checking it against design document
or specification” while system testing “validate a program by checking it against the published user or system requirements”.

Functional testing is the process of determining the speed or effectiveness of a computer, network, software program or device Lin et al (2007). This process can involve quantitative tests done in a lab, such as measuring the response time or the number of MIPS (millions of instructions per second) at which a system functions.

Functional testing typically involves five steps:

1. The creation of input data based on the function’s specifications.
2. The determination of output based on the function’s specifications.
3. The execution of the test case.
4. The identification of functions that the software is expected to perform and
5. The comparison of actual and expected outputs.

1.3.4 Acceptance Testing

Acceptance testing is the final stage of testing that is performed on a system prior to the system being delivered to a live environment. Systems subjected to acceptance testing might include such deliverables as a software system or a mechanical hardware system Zheng et al (2006).
Acceptance tests are generally performed as “black box” tests. Black box testing means that the tester uses specified inputs into the system and verifies that the resulting outputs are correct, without knowledge of the system's internal workings.

The acceptance test suite is run against the supplied input data or using an acceptance test script to direct the testers Xie (2007). Then the results obtained are compared with the expected results. If there is a correct match for every case, the test suite is said to pass. If not, the system may either be rejected or accepted on conditions previously agreed between the sponsor and the manufacturer Keller and Schneidewind (1997).

The objective is to provide confidence that the delivered system meets the business requirements of both sponsors and users. The acceptance phase may also act as the final quality gateway, where any quality defects not previously detected may be uncovered.

1.4 TESTING VERSUS DEBUGGING

Testing and debugging are often lumped under the same heading, and it’s no wonder that their roles are often confused: for some, the two words are synonymous; for others, the phrase “test and debug” is treated as a single word Parnas et al (1990). The purpose of testing is to show that a program has bugs. The purpose of debugging is to find the error or misconception that led to the program’s failure and to design and implement the program changes that correct the error Dai et al (2005). Debugging usually follows testing, but they differ as to goals, methods, and most important, psychology Voas (2003):
1. Testing starts with known conditions, uses predefined procedures, and has predictable outcomes; only whether or not the program passes the test is unpredictable. Debugging starts from possibly un-known initial conditions, and the end cannot be predicted, except statistically.

2. Testing can and should be planned, designed, and scheduled. The procedures for, and duration of, debugging cannot be so constrained.

3. Testing is a demonstration of error or apparent correctness. Debugging is a deductive process.

4. Testing proves a programmer’s failure. Debugging is the programmer’s vindication.

5. Testing, as executed, should strive to be predictable, dull, constrained, rigid and in human. Debugging demands intuitive leaps, conjectures, experimentation, and freedom.

6. Much of testing can be done without design knowledge. Debugging is impossible without detailed design knowledge.

7. An outsider can often do testing. An insider must do debugging.

8. Although there is a robust theory of testing that establishes theoretical limits to what testing can and can’t do, debugging has only recently been attacked by theorists – and so far there are only rudimentary results.
9. Much of test execution and design can be automated. Automated debugging is still a dream.

1.5 SYSTEM RELIABILITY

Assume QP components with constant failure intensities where their reliabilities are measured over a common calendar time interval and that all must function correctly for system success, then the system failure intensity is given by

\[ \lambda = \sum_{k=1}^{Q_k} \lambda_k \]  

(1.1)

where \( \lambda_k \) refers to the individual component failure intensities.

- Software reliabilities are usually given in terms of execution time
- Before software and hardware reliabilities are combined, the conversion of software reliability is required.
- First convert the reliability \( R \) of each software component to failure intensity.
- Using \( \lambda, \tau \) to represent failure intensity as being with respect to execution time, we obtain using \( R(\tau) = \exp(\tau \lambda_{\tau}) \)

\[ \lambda_{\tau} = -\frac{1}{\tau} \ln R \]  

(1.2)

where \( \tau \) is the execution time period for which the reliability was given.
1.6 SOFTWARE RELIABILITY MODEL

In prediction and estimation of software reliability a general method is the use of statistical models Everett (1995). These models make use of either historical data of similar projects or organizations or direct software measures such as fault density, defect density, and defect detection rate of the software under investigation Everett (1995). Some of the well-known examples of software reliability models are Musa’s Execution Time Model Musa et al (1990), Putnam’s Model, Goel-Okumoto Model et al (1979), Generalized Goel NHPP Model Huang et al (2003), Jelinski-Moranda et al (1996), and Littlewood-Verrall Model et al (1995). All these models, as expected, have their own set of advantages and disadvantages that take their roots from their specific assumptions Tausworthe and Lyu (1996). In addition to those model approaches, there exist other techniques for assessment of software reliability. Test coverage techniques, execution path and error seeding are examples of these alternative approaches Malaiya et al (2002).

In the literature different approaches to estimation of the reliability of a software program have been reported Huang et al (2003). The problem with the estimation approach is that it can only be used at later stages of software development process, which channels organizations to use of reliability prediction techniques Huang et al (1999). Software reliability prediction techniques are especially useful when knowledge of approximate reliability level of the software to be developed is desired at early stages of development life cycle. When that information is of critical importance, the performance of prediction process in determination of an initial guess can be improved by the use of more than one prediction model over the same data Brocklehurst and Littlewood (1992).
One of the major problems of software reliability prediction models is that they fail to predict the reliability accurately Lanubile (1996). The reason is that they assume limited historical data of special kind of organizations or of specific type of projects. That creates the problem of loss of control over customization of model’s criteria to fit it to a specific organization Lanubile (1996). Reliability estimation models Wood (1997) can overcome this problem up to some extent. The estimation models are usually in the form of non-homogeneous Poisson processes (NHPP) or Markoff systems Tausworthe and Lyu (1996). Most of the time the difference between the models arises from the definition (or assumption) of “beginning time of the process” or selection of random variable of the model as being either “number of faults detected” or “total number of faults predicted”. In the literature, however, it is possible to come across with models that do not require detection of all the failures Kapur and Garg (1992). Models that relate reliability to cost and priority of failures also exists Boland and Singh (2003).

1.7 PROJECT MANAGEMENT AND RELIABILITY

A direct use of software reliability studies appears in deciding the time when the product is ready to release Yang and Chao (1995). According to the current level of reliability, the amount of necessary testing is determined from the software reliability models by making use of failure data. By this way, it is also possible to measure the cost of certain amount of increase in the reliability in terms of time, budget and man-hours Ehrlich et al (1993). Reference Musa et al (1990) presents a valuable discussion on how software development models affect the overall reliability of a software system. The models investigated in that study are Waterfall Model, Classic Development Model, Prototyping Approach, Spiral Model, Incremental Development Model, and Cleanroom Model.
Among these, Waterfall Model is criticized for not allowing the solution of an inherent problem noticed in later phases, which increases the cost of reliability Boland et al Singh (2003). The problem of Classic Development Model Huang et al (1998) with respect to reliability is stated to be the inefficiency of the model to help customer in determination of requirements in a clear manner. In that study, Prototyping Approach is suggested for improvement of quality and reliability Schneidewind (2003) since it provides feedback from the customer and actual users of the system. It is also indicated that Risk Analysis actions performed in each cycle of Spiral Model contributes to quality and reliability of the software system. According to Musa et al (1990) it is advisable to employ Incremental Development Model if specific functions/modules of the product have more strict reliability requirements.

1.8 ISSUES IN SOFTWARE RELIABILITY MODEL

The work on software reliability models started in 70’s, the first model being presented in 1972. Today the number of existing models exceeds hundred with more models developed every year. Still there does not exist any model that can be applied in all cases Chatterjee et al (1997). Models that are good in general are not always the best choice for a particular data set, and it is not possible to know in advance what model should be used in any particular case Musa et al (1990). Since software reliability models are used in different phases of the Software Development Life Cycle (SDLC), the reliability models are broadly classified under the following categories:
Early prediction models use characteristics of the software development process from requirement to design and test, and extrapolate this information to predict the behavior of software during operation Huang et al (2003). Software reliability growth models (SRGM) captures the failure behavior of software during testing and extrapolates it to determine its behavior during operation. Hence this category of models uses failure data information and trends observed in the failure data to revive reliability prediction Haverkort and Meeuwissen (1995). The SRGMs are further classified as Concave models and Shaped models Brocklehurst and Littlewood (1992). Goel-Okumoto model is one of the most widely used SRGM. In this model, the failure arrival process is assumed to be non-homogeneous Poisson process (NHPP).

In the field of software reliability modeling, the span of time may be considered as calendar time, clock time, and execution time. Musa (1990) and Ohba (1984) showed that the effort index or the execution time is a better time domain for software reliability modeling than the calendar time because the shape of the observed reliability growth curve depends strongly on the time distribution of the testing-effort. Recently, Yamada et al (1986) and Huang et al (1999) proposed a new and simple SRGM which describes the relationship among the calendar testing, the amount of testing-effort, and the number of software faults detected by testing. The test-effort index is measured by the number of CPU hours, the number of test runs, and so on. Unfortunately, most papers assumed that the consumption rate of testing resource expenditures during the testing phase is a constant or even do not consider such testing effort Rameshwar Gupta and Debasis Kundu (2000). In fact, if the effort index data or execution-time-based data are available in the actual observed data set, software reliability models should be developed by
incorporating the testing-effort functions in real development environment (1987, 1999).

- Musa has shown that resource usage is linearly proportional to execution time and mean failures experienced.

- Let $\chi_r$ represent the usage of resource $r$, then

$$\chi_r = \theta_r \tau + \mu_r \mu$$  \hspace{1cm} (1.3)

where $\theta_r$ is the resource usage per CPU hour; $\mu_r$ is the resource usage per failure

- The following table from Musa summarizes the typical parameters

Table1.1 Musa’s Summary of the typical Parameters

<table>
<thead>
<tr>
<th>Resources</th>
<th>Usage per CPU hour</th>
<th>Usage per Failure</th>
<th>Available</th>
<th>Planned Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failure Identification Personnel</td>
<td>$\theta_I$</td>
<td>$\mu_I$</td>
<td>$P_I$</td>
<td>1</td>
</tr>
<tr>
<td>Failure Correction Personnel</td>
<td>0</td>
<td>$\mu_F$</td>
<td>$P_F$</td>
<td>$\rho_F$</td>
</tr>
<tr>
<td>Computer Time</td>
<td>$\theta_C$</td>
<td>$\mu_C$</td>
<td>$P_C$</td>
<td>$\rho_C$</td>
</tr>
</tbody>
</table>
1.9 OUTLINE AND CONTRIBUTIONS

The organization of chapters in the thesis is outlined in the following paragraphs.

Chapter 2 discusses Reliability Engineering. It is an important aspect of many system development efforts and consequently there has been a great deal of research in the software based systems. One important activity included in reliability engineering is reliability prediction and improvement. This chapter also deals with reliability improvement using growth modeling techniques. Software reliability plays a vital role in improvement of quality on software. This chapter proposes two software growth models. This chapter presents improved Bayesian Approach for uncertainty analysis. This chapter presents Software Reliability models based on Logistic Effort Test Function.

Chapter 3 proposes a software quality support tool, a source code evaluator, and a code profiler based on computational intelligence techniques to reduce schedule slippage of development activity. It gives a new approach to evaluate and identify inaccurate source code usage and transitivity, the software product itself. This chapter also proposes the effective software reliability allocation technique that supports multi objective optimization and assists schedule planning as well as effective resources allocation.

Chapter 4 proposes a procedure for analyzing reliable data. Reliability is the indirect measure to improve the quality of the software. Testing is the process of running a system with the intention of finding errors. Testing enhances the integrity of a system by detecting deviations in design and errors in the system. Failure Data used for the testing should be reliable.
Chapter 5 presents a framework that helps to understand about the on-going project and also used to infer the current and future situations in the project or reconstruct the project. The applications of the proposed procedures are explained through two real data sets. The studies show that the proposed simulation procedures can analyze the influence on the performance, and the cost related to software debugging when the number of allocated debuggers changes. This useful, important information can guide project managers in the estimation and adjustment of the staffing needs for debugging systems.

Chapter 6 presents the summary of conclusions of the research along with directions for future research work.