Design, Testing, and Evaluation Techniques for Software Reliability Engineering

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

Abstract

Software reliability is closely influenced by the creation, manifestation and impact of software faults. Consequently, software reliability can be improved by treating software faults properly, using techniques of fault tolerance, fault removal, and fault prediction. Fault tolerance techniques achieve the design for reliability, fault removal techniques achieve the testing for reliability, and fault prediction techniques achieve the evaluation for reliability. In this paper we present best current practices in software reliability engineering for the design, testing, and evaluation purposes. We describe how fault tolerant components can be applied in software applications, how software testing can be conducted to show improvement of software reliability, and how software reliability can be modeled and measured for complex systems. We also examine the associated tools for applying these techniques.

1. Introduction

While the advancement for computer hardware has made excellent progress in the past 30 years, proper development of software technology has failed to keep pace in all measures, including quality, productivity, cost, and performance. Software has become the bottleneck of system development, and its delay and cost overrun have often put modern complex projects in jeopardy. With the last decade of the 20th century, computer software has already become the major source of reported outages in many systems [1].

As an example, Figure 1 shows the causes of total outage incidents of U.S. switching systems in 1992, in which we can see that software accounts for 81% of network outages (including Retrofits, Scheduled Events, Software Design, Procedural). Hardware and other faults were only responsible for less than 20% of the outage [2]. Moreover, severe software failures have impaired several high-visibility programs worldwide. These critical incidents either caused enormous revenue losses to companies, or put human lives in danger.

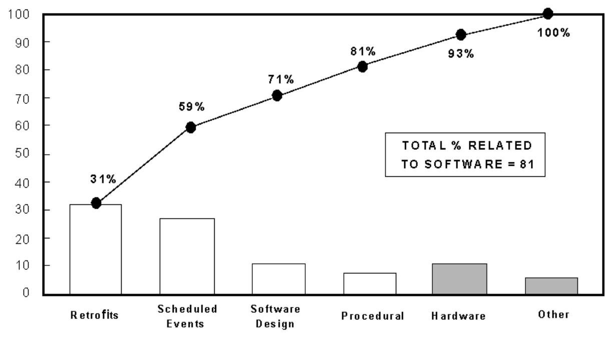


Figure 1. Switching system outage causal classification.

To this end, many software companies see a major share of project development costs identified with the design, implementation, and assurance of reliable software, and they recognize a tremendous need for systematic approaches to assure software reliability within a system. Clearly, developing the required techniques for software reliability engineering is a major challenge to computer engineers, software engineers, and engineers of various disciplines for now and the decades to come.

2. Software Reliability Engineering Overview

Software reliability engineering is centered around a very important software attribute: *reliability*. Software reliability is defined as the probability of failure-free software operation for a specified period of time in a specified environment [3]. It is one of the attributes of software quality, a multi-dimensional property including other factors like functionality, usability, performance, serviceability, capability, installability, maintainability, and documentation [4]. Software reliability, however, is generally accepted as the key factor in software quality, since it is aimed at quantifying and predicting software failures, the unwanted events which can make a powerful system inoperative or even deadly. Thus reliability is an essential ingredient in customer satisfaction for most commercial companies and governmental organizations. In fact, ISO 9000-3 specifies measurement of field failures as the only required quality metric: "... at a minimum, some metrics should be used which represent reported field failures and/or defects form the customer's viewpoint. ... The supplier of software products should collect and act on quantitative measures of the quality of these software products." (See the Section 6.4.1 of [5]).

Reliability engineering is a daily practiced technique in many engineering disciplines. Using a similar concept in these disciplines, we define *software reliability engineering* as the quantitative study of the operational behavior of software-based systems with respect to user requirements concerning reliability. Software reliability engineering therefore includes [6]:

(1) software reliability measurement, which includes estimation and prediction, with the help of software reliability models established in the literature;
(2) the attributes and metrics of product design, development process, system architecture, software operational environment, and their implications on reliability; and
(3) the application of this knowledge in specifying and guiding system software architecture, development, testing, acquisition, use, and maintenance.

In this paper we discuss software reliability engineering techniques in three categories: (1) *design* for software reliability, (2) *testing* for software reliability, and (3) *evaluation* for software reliability. In the design category, reliability of the software system is achieved by developing reliable components for the system. The key is to provide *fault avoidance* and *fault tolerance*. The available techniques we emphasize include reusable software fault tolerance routines, and software fault tolerance by design diversity. In the testing category, reliability of the software system is improved by testing techniques. The key topic is to provide *fault removal*. The available techniques include data flow testing, fault injection testing, and the associated tools. In the evaluation category, reliability of software is demonstrated by modeling techniques. The key topic is to provide *fault prediction*. The available techniques include software reliability measurement tasks and software reliability tools. We discuss the details of these techniques in the following three sections.

3. Design for Software Reliability

Design for reliability is aimed at achieving reliability of the software system under development, using fault avoidance and fault tolerance techniques. Fault avoidance is addressed by many software engineering techniques and is beyond the scope of this paper. Fault tolerance, on the other hand, is the focus of our discussion. We examine fault tolerance techniques used in single-version as well as multiple-version environments.

3.1 Single-Version Software Fault Tolerance

Software fault tolerance in single-version software environment is achieved by introducing special fault detection and recovery features, including modularity, system closure, atomic actions, decision verification, and exception handling. One successful approach is accomplished by reusable routines for software fault tolerance [7].

Traditionally, reliability is provided through fault tolerance technology in the hardware, operating system and database layers of a computer system executing the application software. Two trends are emerging in the marketplace. First, standard commercial hardware and operating systems are becoming more reliable, distributed, and inexpensive. They are now off-the-shelf, commodity items with open and evolving standards and interfaces. Second, the proportion of failures due to faults in the *application software* is increasing due to increased size and complexity of software being deployed.

To implement application-level software fault tolerance, we need a mechanism to *detect* and *restart* failed processes at the minimum. The next higher level is to perform *checkpoint* and *recovery* for the internal state of a process when it fails. Additionally, logging and replaying messages may also be employed. It may happen that some part of the environment will change during recovery and replay in a way that the process will not fail upon re-execution. Another method is to reorder the messages during replay so that errors due to unexpected event sequences are masked. The next higher level is *on-line replication* of application files at a remote site in addition to the previous tasks.

In addition to the reactive recovery procedures described above, there is a complementary proactive approach, called *software rejuvenation*, to handle transient software errors. Software rejuvenation prevents failures from occurring by periodically and gracefully terminating an application and immediately restarting it at a clean internal state. Restarting an application involves queuing the incoming messages, re-spawning the application processes at an initial state, reinitializing the in-memory volatile data structures, and logging administrative records.

Figure 2 shows a middleware platform, Software Implemented Fault Tolerance (SwIFT), which includes a set of reusable software components (*watchd*, *libft*, *REPL*, *libckp*, and *addrejuv*) to perform software fault tolerance schemes. The hardware platform is a network of standard computers where each computer provides a back-up facility for another one on the network. The components provide mechanisms to checkpoint data, log messages, watch and detect errors, rollback and restart processes, recover from failures, and rejuvenate to avoid failures.

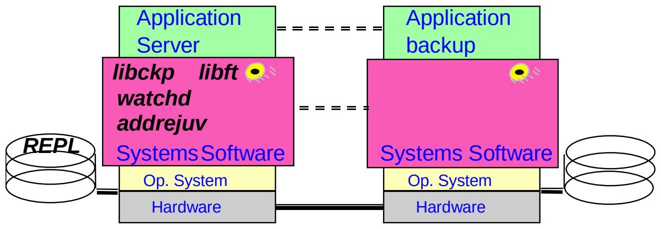


Figure 2. SwIFT platform and components.

Watchd is a watchdog daemon process that runs on a single machine or on a network of machines, whose purpose is to detect application process failures and machine crashes. It determines whether a process is hung by either polling the application or checking an "I-Am-Alive" heartbeat message periodically sent from the application process to *watchd*. When *watchd* detects that an application process crashed or failed, it recovers that application at an initial internal state or at the last checkpointed state. It is recovered on the primary node if that node has not crashed, otherwise on the backup node for the primary as specified in a configuration file.

Libft is a user-level library that can be used in application programs to specify and checkpoint critical data, recover the checkpointed data, log events, locate and reconnect to a backup server. It provides a set of functions to specify critical volatile data (i.e., data in the memory) in an application. These critical data items are allocated in a reserved region of the virtual memory and are periodically checkpointed on primary and backup nodes.

REPL is a file replication mechanism for on-line replication of critical files of an application. The mechanism uses dynamic-shared libraries to intercept file system calls for data replication in a remote site. *REPL* is built on top of standard file systems, requiring no change to the underlying operating system. Speed, robustness and replication transparency are the primary design goals of the *REPL* replication mechanism.

Libckp is a user-transparent checkpointing library. It can be linked with a user's program to periodically save the program state on stable storage (e.g., disks) without requiring any modification to the source code. When a process rolls back, all the modifications it has made to external files since the last checkpoint are undone so that the states of the files are consistent with the checkpointed state. *Libckp* also provides application-initiated checkpoint and rollback facilities within a program. This facilitates restoration of global/static variables, dynamically allocated memory, and user files.

Addrejuv is an added feature of *watchd* to do software rejuvenation by stopping and restarting a process at a certain interval or when a particular event happens in the application process. The interval or event for periodic rejuvenation is determined through analysis and experience with the application [8]. When the *addrejuv* feature is used, *watchd* creates a rejuvenation shell script and registers the starting time or the event for execution of that script with a system daemon to rejuvenate the process. The shell script takes systematic steps to stop the process. Once the process is terminated, *watchd* takes a recovery action to re-spawn the process in the same manner as it does when it detects a failure.

3.2 Multiple-Version Software Fault Tolerance.

The evolution of using design diversity [9] techniques for building fault-tolerant software out of simplex units has taken two directions: *N-version software* (NVS) shown in Figure 3 and *recovery blocks* (RB) shown in Figure 4.

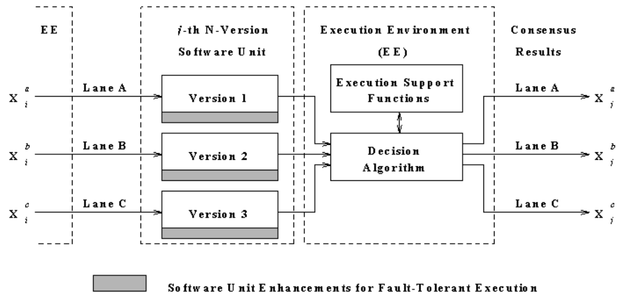

EE
i-th N-Version Software Unit
Execution Environment (EE)
Consensus Results
Lane A
Lane B
Lane C
Version 1
Version 2
Version 3
Execution Support Functions
Decision Algorithm
Lane A
Lane B
Lane C
Software Unit Enhancements for Fault-Tolerant Execution

Figure 3. The *N*-version software (NVS) model with *N* = 3

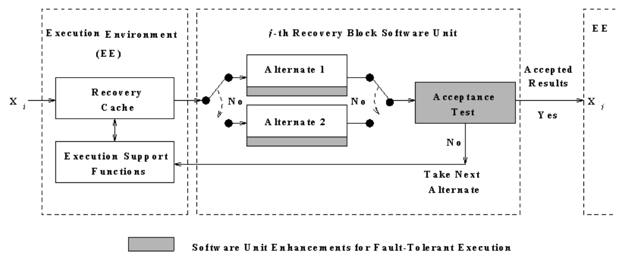
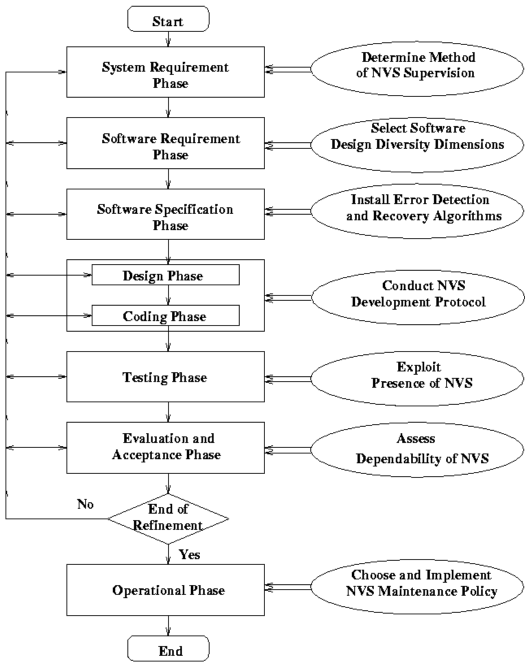

Execution Environment (EE)
i-th Recovery Block Software Unit
EE
Recovery Cache
Execution Support Functions
Alternate 1
Alternate 2
No
No
Acceptance Test
Accepted Results
Yes
No
Take Next Alternate
Software Unit Enhancements for Fault-Tolerant Execution

Figure 4. The recovery block (RB) model.

The common property of both schemes is that two or more diverse units (called *versions* in NVS, and *alternates* and *acceptance tests* in RB) are employed to form a fault-tolerant software unit. The most fundamental difference is the method by which the decision is made that determines the outputs from the fault-tolerant system. The NVS approach employs a generic *decision algorithm* that is provided by the *execution environment* (EE) and looks for a *consensus* of two or more outputs among *N* member versions. The RB model applies the *acceptance test* to the output of an individual alternate; this acceptance test must be *specific* for every distinct service, i.e., it is customdesigned for a given application, and is a member of the RB fault-tolerant software unit, but not a part of the EE.

Both RB and NVS have evolved procedures for error recovery. In RB, backward recovery is achieved in a hierarchical manner through a *nesting* of RBs, supported by a *recovery cache* that is part of the EE. In NVS, forward recovery is done by the use of the *community error recovery* algorithm that is supported by the specification of *recovery points* and by the decision algorithm of the EE. Both recovery methods have limitations: in RB, errors that are not detected by an acceptance test are passed along and do not trigger recovery; in NVS, recovery will fail if a majority of versions have the same erroneous state at the recovery point.

The procedure to develop diversified software units for RB and NVS is formulated in an *N*-version programming (NVP) design paradigm [10], as shown in Figure 5. The purpose of the paradigm is to integrate the unique requirements of NVP with the conventional steps of software development methodology. The application of a proven software development method is the foundation of the NVP paradigm. This method is supplemented by procedures that aim: (1) to attain suitable isolation and independence (with respect to software faults) of the *N* concurrent version development efforts, (2) to encourage potential diversity among the *N* versions of an *N*-version software unit, and (3) to elaborate efficient error detection and recovery mechanisms.



Start
System Requirement Phase
Determine Method of NVS Supervision
Software Requirement Phase
Select Software Design Diversity Dimensions
Software Specification Phase
Install Error Detection and Recovery Algorithms
Design Phase
Coding Phase
Conduct NVS Development Protocol
Testing Phase
Exploit Presence of NVS
Evaluation and Acceptance Phase
Assess Dependability of NVS
No
End of Refinement
Yes
Operational Phase
Choose and Implement NVS Maintenance Policy
End

Figure 5. A design paradigm for NVP.

4. Testing for Software Reliability

The goal for software testing is to improve software reliability by removing faults before system operational phase. We examine data flow testing for general systems, and fault insertion testing for fault-tolerant systems.

4.1 Software Coverage Testing Scheme and Tool

There are many ways of testing software. The terms *functional*, *regression*, *integration*, *product*, *unit*, *coverage*, *user-oriented*, are only a few of the characterizations we encounter. These terms are derived from the method of software testing or the development phase during which the software is tested. The testing methods "functional," "coverage," and "user-oriented," indicate, respectively, that the *functionality*, the *structure*, and the *user view* of the software are to be tested. Any of these methods might be applied during the unit, integration, product, or regression phases of the software's development.

White-box, or *coverage*, testing uses the structure of the software to measure the quality of testing. This structural coverage and its measurement is considered to be connected with reliability estimation. These testing schemes include statement coverage testing, decision coverage testing, and data-flow coverage testing.

Statement coverage testing directs the tester to construct test cases such that each statement or a basic block of code, is executed at least once.

Decision coverage testing directs the tester to construct test cases such that each decision in the program is covered at least once.

Data flow coverage testing directs the tester to construct test cases such that all the def-use pairs are covered. Consider a statement $S_1{:}x=f()$ in program *P*, where *f* is an arbitrary function. Let there be another statement $S_2{:}p=g(x,*)$ in *P* where *g* is an arbitrary function of *x* and any other program variables. We say that S_1 is a definition and S_2 a use of the variable *x*. The two occurrences of *x* constitute a def-use pair.

If the use of a variable appears in a computational expression, then such a pair is termed as a c-use. If the use appears inside a predicate then the pair is termed as a p-use.

Coverage measures from the above testing criteria are obtainable from the ATAC tool. ATAC (Automatic Test Analysis for C) is a software testing tool for the measurement of data flow coverage for C programs during their execution [11]. Using ATAC, we show the relationship between testing and reliability using two real-world applications. The first application is an automatic (i.e., computerized) airplane landing system, or so-called *autopilot*, developed and programmed by 15 programming teams at the University of Iowa and the Rockwell/Collins Avionics Division [12], using the NVP design paradigm described in Figure 5. 12 versions of the autopilot program were produced and accepted at the end of the project. The coverage measures obtained from this project and the fault detection history are depicted in Figure 6.

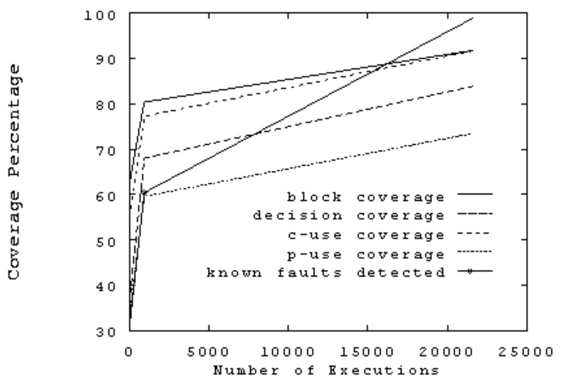


Figure 6. Relationship between coverage improvement and fault detection during testing phases.

Figure 6 shows the progress of software testing from unit testing (1 complete flight simulation test case), integration testing (960 test cases), to acceptance testing (21600 test cases). The dash lines depict the accumulation of test coverage, while the solid line depicts the increased percentage of fault detection. The data points are taken from the average of the resulting 12 programs. It can be seen from Figure 6 that as the number of program executions increases, the data flow coverage increases, and the number of detected faults also increases. Both the coverage and the detected faults, however, do not increase linearly with respect to the number of program executions.

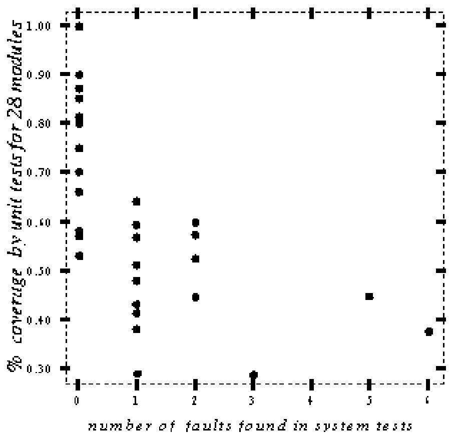


Figure 7. Relationship of unit coverage testing to system test faults for one system

Figure 7 displays data from another experiment to compare the statement coverage of unit tests for 28 modules of a single system to the number of system test faults found for each module [13]. From this figure, again, we can see a clear relationship between high statement coverage in unit testing and low number of faults detected in system test.

4.2 Software Fault Insertion Testing

The main objective of Software Fault Insertion Testing (SFIT) is to test fault tolerance capability through injecting faults into the system and analyze if the system can detect and recover from various fault scenarios. With SFIT, several network-wide system outages could be avoided.

SFIT is recommended for system testing or acceptance testing during the testing life cycle. In this way, the system's overall reaction to faults/errors/failures can be observed and analyzed. However, in some cases, SFIT can also be performed at the unit testing level where the fault manager functionality resides in a local subsystem level.

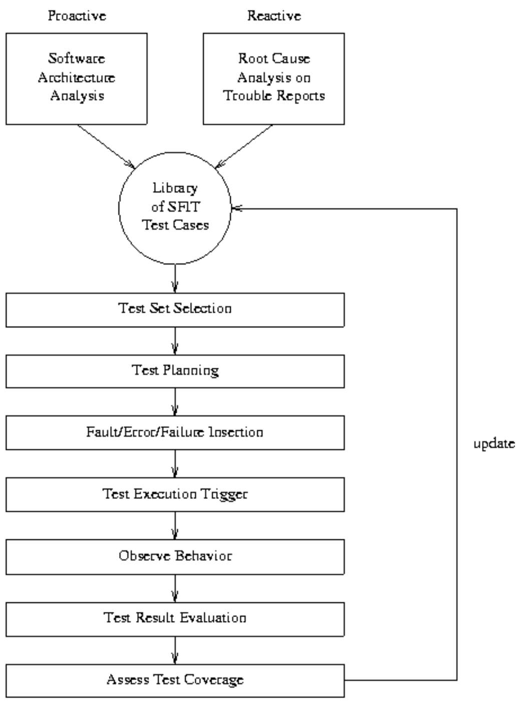


Figure 8: Software Fault Insertion Testing Methodology

Figure 8 shows the methodology used for SFIT [14]. This methodology consists of the following steps:

1. Software Architecture Analysis. Before conducting SFIT, a sufficient knowledge of the software (including functions of some key software components, such as error recovery software subsystem) and its architecture is needed. This

analysis is proactive and consists of three key parts: application software analysis, fault manager analysis, and interface analysis.

2. Root Cause Analysis. Internal testing results and external field problems often can give a good indication of the product's reliability. Therefore, root cause analysis on the internal trouble reports and customer service reports can help the testing organization to identify common problems that need to be addressed in SFIT. Root cause analysis is more reactive; nevertheless, it can help to identify the area and type of faults/errors to be tested.
3. Test Set Selection. During the test set selection, the following two aspects need to be identified: (a) properties/predicates to be checked for assessing the expected behavior of the system in the presence of the injected faults, and (b) observations to be made to verify the assertion of the corresponding actual system behavior.
4. Test Planning. After the test set has been selected, testing needs to be planned. For example, test scripts need to be prepared based on the selected test set. For code-based injection, software patches need to be prepared in advance. For state-based injection, appropriate tools need to be allocated to change the state of the system.
5. Fault/Error Insertion. With all the test cases available, this step involves the actual insertion of faults in the code or errors in the state. The location of faults or errors should be identified during this step.
6. Test Execution Trigger. A fault in the system may not be activated when it is inserted into the system. Therefore, the test trigger needs to be set during this step to activate the inserted faults. Triggers could be input values from the users, internal and external events, or messages.
7. Observe Behavior. This step observes the system reaction to the inserted faults or errors within a specified time frame.
8. Test Result Evaluation. The test result can reveal the effectiveness of the test cases as well as weakness of the system's fault tolerance capability. The test result evaluation step can help to eliminate less effective test cases and identify areas for improvements for the system's fault tolerance mechanism.
9. Assess Test Coverage. Although complete testing coverage with SFIT is not economically possible, a notion of test coverage adequacy is essential to the confidence in the fault tolerance of a system.
10. Update SFIT Test Case Library. A library of common and generic faults, errors, and failures along with their attributes (such as frequency of occurrence or severity) should be collected and stored. This library can be used to define test input for fault tolerance testing. In addition, faults designed to test for the rare, unusual, and severe fault tolerance conditions of the system will be added to the repository. By this, an adequacy criterion for fault insertion testing can be gradually established.

5. Evaluation for Software Reliability

Evaluation for reliability is focused on the modeling and analysis techniques for fault prediction purpose. We discuss a systematic software reliability measurement procedure, and a software reliability estimation tool.

5.1 Software Reliability Measurement Procedure

Software reliability measurement is the application of statistical inference procedures to failure data taken from software testing and operation to determine software reliability. We have established a framework for software reliability measurement purpose, as described in Figure 9.

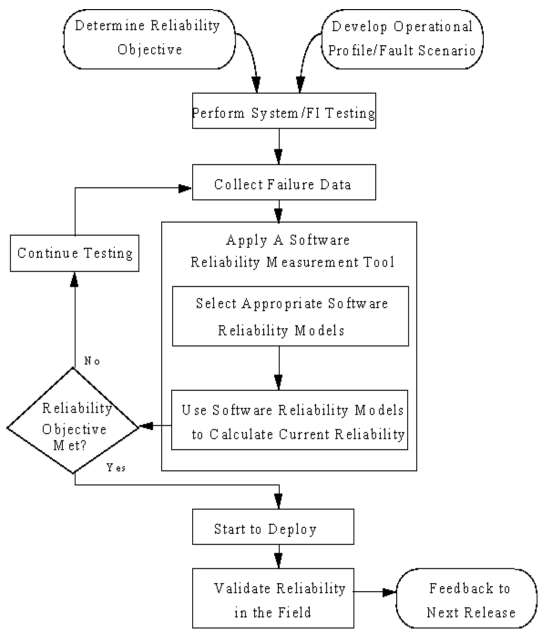

Determine Reliability Objective
Develop Operational Profile/Fault Scenario
Perform System/FI Testing
Collect Failure Data
Continue Testing
Apply A Software Reliability Measurement Tool
Select Appropriate Software Reliability Models
Use Software Reliability Models to Calculate Current Reliability
No
Reliability Objective Met?
Yes
Start to Deploy
Validate Reliability in the Field
Feedback to Next Release

Figure 9. Software reliability measurement procedure overview.

It can be seen from Figure 9 that there are four major components in this software reliability measurement process, namely,

(1) *reliability objective,*
(2) *operational profile,*
(3) *reliability modeling and measurement*, and
(4) *reliability validation*.

According to this framework, quality is first defined quantitatively from the customer's viewpoint by defining failures and failure severity, by determining a reliability objective, and by specifying balance among key quality objectives (e.g., reliability, delivery date, cost). Second, customer usage is quantified by developing an operational profile. Operational profile is a set of disjoint alternatives of system operation and their associated probabilities of occurrence (see Chapter 5 in [6]). The construction of an operational profile encourages testers to select test cases according to the system's operational usage, which contributes to more accurate estimation of software reliability in the field. In this procedure, quality objectives and operational profile are employed to manage resources and to guide design, implementation, and testing of software. Moreover, reliability during testing is tracked to determine product release, using appropriate software reliability measurement models and tools. This activity may be repeated until a certain reliability level has been achieved. Finally, reliability can be analyzed in the field to validate the reliability engineering effort and to provide feedback for product and process improvements.

Reliability modeling is an essential element of the reliability estimation process. It determines if a product meets its reliability objective and is ready for release. It is required to use a reliability model to calculate, from failure data collected during system testing (such as failure report data and test time), various estimates of a product's reliability as a function of test time. These reliability estimates can provide the following information 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.
(4) The predicted reliability beyond the system testing already performed.

5.2 Software Reliability Measurement Tool

There are as many as 40 software reliability models proposed in the literature. Despite the existence a large quantity (and variation) of these models, the problem of model selection and application is manageable.

Using the statistical methods provide in [6] (Chapter 4), "best" estimates of reliability can be obtained during testing. These estimates are then used to project the reliability during field operation in order to determine if the reliability objective has been met. This procedure is an iterative process since more testing will be needed if the objective is not met.

Since the engagement and application of software reliability models and the evaluation and interpretation of model results involve tedious computation-intensive tasks, we believe the only practical usage of reliability models is through software tools. For this purpose, we designed and implemented a software reliability modeling tool, called Computer-Aided Software Reliability Estimation (CASRE) system [15], for an automatic and systematic approach in estimating software reliability.

CASRE is implemented as a software reliability modeling tool that addresses the ease-of-use issue as well as other issues. Figure 10 shows the high-level architecture for CASRE.

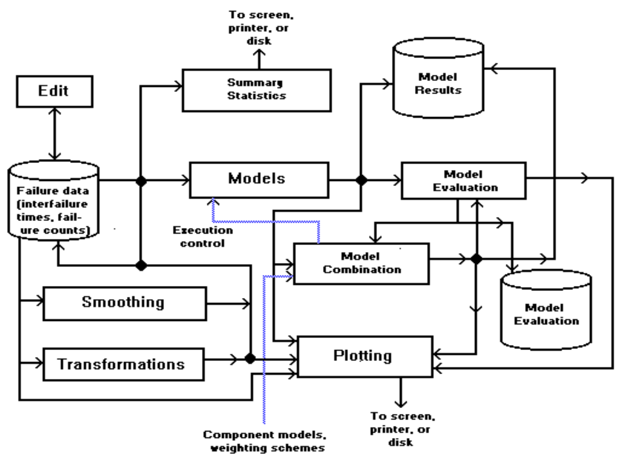


Figure 10. High-level architecture for CASRE.

CASRE is designed for the Windows environment. A Web-based version is also available [17]. The command interface is menu driven; users are guided through the selecting of a set of failure data and executing a model by selectively enabling pull-down menu options. Modeling results are also presented in a graphical manner. Users can select multiple models from two categories depending on failure data format: Time-Between-Failures models (for inter-failure times) or Failure-Count models (for failure intensities).

After one or more models have been executed, the predicted failure intensities or inter-failure times are drawn in a graphical display window. Users can manipulate this window's controls to display the results in a variety of ways, including cumulative number of failures and the reliability growth curve. Users may also display the results in a tabular fashion if they wish. The performance of each model is evaluated using multiple criteria to assess model accuracy, model bias, model bias trend, and model noise. Based on these criteria, the best model or models can be selected for reliable prediction of the software reliability. In addition, CASRE is facilitated with a useful functionality where results from different models can be combined in various ways to yield reliability estimates whose predictive quality is better than that of the individual models themselves [16].

CASRE has been used by major corporations including AT&T, Lucent, Microsoft, NASA, IBM, Motorola, Nortel, etc. It is available through NASA Cosmic software distribution center, and a software diskette in [6].

6. Conclusions

Developing reliable software systems is a formidable task, which involve the best of our knowledge in software reliability techniques. This paper surveys the current schemes in the design, testing, and evaluation of software reliability. We describe the reliability techniques associated with each of these three activities for fault tolerance, fault removal, and fault prediction. We also discuss the available software tools, and some project application results.

References

[1] J. Gray, “A Census of Tandem System Availability Between 1985 and 1990,” *IEEE Transactions on Reliability* 39(4):409-418, October 1990.

[2] National Reliability Council (NRC) Switch Focus Team Report, June 1993.

[3] Institute of Electrical and Electronics Engineers, *ANSI/IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std. 729-1991, 1991.

[4] R.B. Grady, *Practical Software Metrics for Project Management and Process Improvement*, Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[5] International Standard Organization, “Quality Management and Quality Assurance Standards - Part 3: Guidelines for the Application of ISO 9001 to the Development, Supply and Maintenance of Software,” *ISO 9000-3*, Switzerland, June 1991.

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

[7] Y. Huang, C.M.R. Kintala, L. Bernstein, and Y.-M. Wang, “Components for Software Fault Tolerance and Rejuvenation,” *AT&T Technical Journal*, 29-37, March/Spril 1996.

[8] Y. Huang, C.M.R. Kintala, N. Kolettis, and N.D. Fulton, “Software Rejuvenation: Analysis, Module and Applications,” *Proceedings of 25th International Symposium on Fault-Tolerant Computing* (FTCS-25), 381-390, Pasadena, California, June 1995.

[9] A. Avizienis, “The Methodology of N-Version Programming,” Chapter 2 of *Software Fault Tolerance*, M. R. Lyu (ed.), Wiley, 23-46, 1995.

[10] M.R. Lyu, “A Design Paradigm for Multi-Version Software,” *Ph.D. Dissertation*, UCLA, Computer Science Department, May 1988.

[11] M.R. Lyu, J.R. Horgan, and S. London, “A Coverage Analysis Tool for the Effectiveness of Software Testing,” IEEE Transactions on Reliability, 43(4):527-535, December 1994.

[12] M.R. Lyu and Y. He, “Improving the N-Version Programming Process Through the Evolution of a Design Paradigm,” *IEEE Transactions on Reliability*, 42(2):179 - 189, June 1993.

[13] S.R. Dalal, J.R. Horgan, and J.R. Kettenring, “Reliable Software and Communication: Software Quality, Reliability, and Safety,” *Proceedings of the 15th International Conference on Software Engineering*, Baltimore, MD, May 1993.

[14] M.Y. Lai and S.Y. Wang, "Software Fault Insertion Testing for Fault Tolerance," Chapter 13 of Software Fault Tolerance, M.R. Lyu (ed.), Wiley, 315-333, 1995.

[15] M.R. Lyu and A. Nikora, “CASRE - A Computer-Aided Software Reliability Estimation Tool,*” Proceedings of Computer-Aided Software Engineering Workshop*, 264-275, Montreal, Canada, July 1992.

[16] M.R. Lyu and A. Nikora, “Using Software Reliability Models More Effectively,” *IEEE Software*, 43-52, July 1992.

[17] M.R. Lyu and Juergen Schoenwaelder, “Web-CASRE: A Web-Based Tool for Software Reliability Measurement,” Proceedings of International Symposium on Software Reliability Engineering, Paderborn, Germany, November, 1998.