An Integrated Approach to Achieving High Software Reliability

Michael R. Lyu
Bell Laboratories, Lucent Technologies
600 Mountain Avenue
Murray Hill, NJ 07974
908-582-5366
lyu@research.bell-labs.com

Abstract— In this paper we address the development, testing, and evaluation schemes for software reliability, and the integration of these schemes into a unified and consistent paradigm. Specifically, techniques and tools for the three phases of software reliability engineering will be described. The three phases are (1) modeling and analysis, (2) design and implementation, and (3) testing and measurement.

In the modeling and analysis phase we describe Markov modeling and fault-tree analysis techniques. We present system-level reliability models based on these techniques, and provide modeling examples for the reliability analysis and study with known system architectures. We describe how reliability block diagrams can be constructed for a real-world system for reliability prediction, and how critical components can be identified from the existing architecture. We also apply fault tree models to fault tolerant system architectures, and formulate the resulting reliability quantity. Finally, we describe two software tools, SHARPE and UltraSAN, which are available for reliability modeling and analysis purpose.

In the design and implementation phase we show specific fault-tolerant techniques in building reliable software systems for either single-version software or multiple-version software. In single-version software we form a generic platform and a set of reusable software components to perform software fault tolerance tasks in any application executing on that platform. These software fault tolerance components, including watchd, libft, REPL, libckp, and addrejuv, provide a powerful set of building blocks to defend against software faults in various levels of a system. We describe the concept and implementation of these techniques. In addition, we examine multiple-version systems using design diversity, including recovery blocks and N-version programming techniques.

In the testing and measurement phase we describe several software testing schemes, particularly including data flow testing, and software reliability measurement procedures.
We describe the software testing schemes in terms of their effectiveness and their relationship to reliability, as well as provide quantitative comparison between testing coverage and reliability measure. Furthermore, we will provide an in-depth discussion on the software reliability modeling and measurement techniques, including their concepts, approaches, and procedures. In particular, the CASRE tool for automatic reliability measurement will be described and presented. The CASRE system, a computer-aided software reliability estimation tool, is implemented to encapsulate many software reliability modeling techniques in a comprehensive framework via a systematic procedure, and is currently widely distributed in industry.

TABLE OF CONTENTS

1. INTRODUCTION

Our demand for complex hardware/software systems has increased more rapidly than our ability to design, implement, test, and maintain them. When the requirements for and dependencies on computers increase, the crises of computer failures also increases. The impact of these failures ranges from inconvenience (e.g., malfunctions of home appliances), economic damage (e.g., interruptions of banking systems), to loss of life (e.g., failures of flight systems or medical software). The reliability of computer systems has become a major concern for our society. Within the computer revolution progress has been uneven: software assumes a larger burden while based on a less firm foundation than hardware. In stark contrast with the rapid advancement of hardware technology, proper development of software technology has failed to keep pace in all measures, including quality, productivity, cost, and performance. 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.

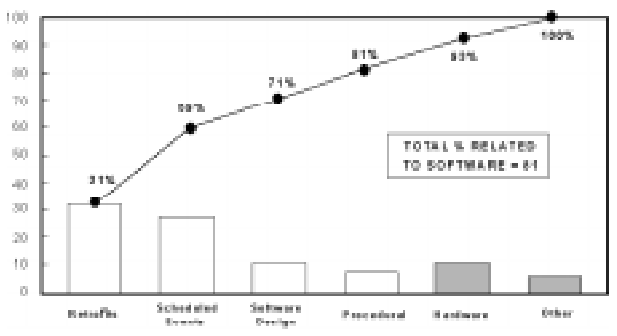

Figure 1 Switching System Outage Causal Classification

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 using software reliability engineering techniques. 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. Phase-Based Approach: An 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 customer satisfaction 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 quantifies software failures - which can make a powerful system inoperative or even deadly. As a result, 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. Civil engineers use it to build bridges and computer hardware engineers use it to design chips and computers. Using a similar concept in these disciplines, we define *software reliability engineering* as \f2the 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 attack the problem of software reliability engineering in three phases: (1) Modeling and Analysis Phase, (2) Design and Implementation Phase, and (3) Testing and Measurement Phase. All these phases deal with the management of software faults and failures. In the Modeling and Analysis Phase, reliability of the software system is being modeled according to the structure of the system and possible fault scenarios. The key topic of this phase is to provide fault modeling of the system, and ask the "what if" questions. The available modeling approaches include system reliability modeling block diagrams, reliability models by Markov chains, fault tree analysis, and stochastic Petri-nets. In the Design and Implementation Phase, reliability of the software system is being achieved by software engineering techniques. The key topic of this phase 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 and Measurement Phase, reliability of the software system is being evaluated and verified by testing and measurement techniques. The key topic of this phase is to provide fault removal and fault prediction. The available techniques include data flow testing, reliability measurement tasks and software reliability tools. We discuss the details of these techniques in the following three sections.

3. Phase 1: Modeling and Analysis Phase

To provide reliability modeling and analysis of a software system during the pre-design phase, the overall system architecture based on requirement can be modeled by several techniques. The available modeling approaches include system reliability modeling block diagrams, Makov-chains reliability modeling, fault tree analysis, and stochastic Petri-nets. These approaches can be used to establish system reliability and performance model for the study of system behavior under various scenarios. The reliability of the system, for example, can be predicted in a coarse basis for the overall system given its known architectural options. Sensitivity analysis can then be performed to locate important parameters of the system, and critical components of the system can be identified for enforcement of each component's individual reliability. Note that the reliability model established in this phase can be refined and revised for evaluation purpose in a post-design phase for a fine prediction and estimation purpose.

3.1 Reliability Block Diagram

Figure 2 shows an example for the reliability modeling and analysis using block diagrams. This is a military distributed processing system which has an mean time to failure (MTTF) requirement of 100 hours and an availability requirement of 0.99. The overall architecture of the system depicted in Figure 2 indicates that the system consists of three subsystems, SYS1, SYS2, SYS3, a local area network, LAN, and a 10 KW power generator GEN. In order for the system to work, all the components (except SYS2) have to work. In the early phase of system testing, hardware reliability parameters are predicted according to the MIL-HDBK-217, and shown for each system component. Namely, above each component block in Figure 2 two numbers appear. The upper number represents the predicted MTTF for that component, and the lower number represents its mean time to repair (MTTR). The units are hours. For example, SYS1 has 280 hours for MTTF and 0.53 hours for MTTR, while SYS2 and SYS3 have 387 hours for MTTF and 0.50 hours for MTTR. Note that SYS2 is configured as a triple module redundant system, shown in the dotted-line block, where the subsystem will work as long as two or more modules work. Due to this fault-tolerant capability, its MTTF improves to 5.01 ? 10^4 hours and MTTR becomes 0.25 hours.

Figure 2 An Example of Predicting System Reliability

To calculate the overall system reliability, all the components in the system have to be considered. If we assume the software does not fail (a mistake often made by system reliability engineers!), the resulting system MTTF would be 125.9 hours, and MTTR would be 0.62 hours, achieving system availability of 0.995. It looks as if the system already meets its original requirements.

But the software does fail. Both SYS2 and SYS3 software contain 300,000 lines of source code, and following the prediction model described in [7], the predicted initial failure rates for SYS2 software and SYS3 software are both 2.52 failures per execution hour. (Note the three SYS2 S/W are identical software copies and not fault-tolerant.) Even without considering SYS1 software failures, the system MTTF would have become 11.9 CPU minutes. If assuming MTTR is still 0.62 hours, the system availability becomes 0.24, far less than it was predicted assuming no software failures.

3.2 Fault Tree Analysis

Fault tree models have long been used for the qualitative and quantitative analysis of the failure modes of critical systems [8]. A fault tree provides a mathematical and graphical representation of the combinations of events which can lead to system failure. The construction of a fault tree model can provide insight into the system by illuminating potential weaknesses with respect to reliability or safety. A fault tree can help with the diagnosis of failure symptoms by illustrating which combinations of events could lead to the observed failure symptoms. The quantitative analysis of a fault tree is used to determine the probability of system failure, given the probability of occurrence for failure events.

The construction of a fault tree, if performed manually, provides a systematic method for analyzing and documenting the potential causes of system failure. The analyst begins with the failure scenario being considered, and decomposes the failure symptom into its possible causes. Each possible cause is then investigated and further refined until the basic causes of the failure are understood. From a system design perspective, the fault tree analysis provides a logical framework for understanding the ways in which a system can fail, which is often as important as understanding how a system can succeed.

A fault tree consists of the undesired top event (system or subsystem failure) linked to more basic events by logic gates. The top event is resolved into its constituent causes, connected by *AND*, *OR* and *M-out-of-N* logic gates, which are then further resolved until basic events are identified. The basic events represent basic causes for the failure, and represent the limit of resolution of the fault tree.
Fault trees do not generally use the {\em NOT} gate, because the inclusion of inversion may lead to a non-coherent fault tree, which complicates analysis. It is quite rare to have need for complementation in a fault tree, so this limitation is acceptable for the analysis of practical systems.

Figure 3 describes an example for applying fault tree analysis to fault-tolerant software (See Section 4), specifically, Distributed Recovery Block (DRB) [9]. The top portion of Figure 3 shows the Markov model of system structure, where the hardware and error confinement areas [10] associated with the DRB architecture are considered. The system is defined by two software variants and two hardware replications. The hardware error confinement area (HECA) is the lightly shaded region, the software error confinement area (SECA) is the darkly shaded region. The HECA or SECA covers the region of the system affected by faults in that component. For example, the HECA covers the software component since the software component will fail if that hardware experiences a fault. It can be seen that originally the system is running on a full configuration with two hardware components and two software components. Upon a hardware failure (with failure rate ? ?and coverage factor *c*), the system can be reconfigured to a degraded configuration with two software variants running on one hardware component. If this hardware failure is not recoverable or if a second hardware failure happens, then the system goes to the failure state.

The middle and lower portions of Figure 3 show how fault tree models can be constructed for the initial and degraded configurations, respectively, for the computation errors. For the initial state, a single task computation will produce unacceptable results if one of three events occur. First, if both the primary and secondary fail on the same input, because of two unrelated faults or a single related fault. Second, if both hardware components experience faults, then the computations being hosted will be upset and be unable to produce correct results. Third, if the decider (acceptance test) fails to either detect unacceptable results or to accept correct results, then the computation fails. Fault tree model for the intermediate state after one hardware failure is the same except that there is only one hardware component left.

The fault tree model provides a compact format for describing the effects of both software and hardware faults. For example, we can easily visualize the effects of a decider failure or a related fault between the versions. To formulate the system behavior quantitatively, we use the following notation for basic events in the fault tree model:

{bf V#} (where \# is an integer between 1 and 4) For (up to) four versions of software, the input for a single computation activates an unrelated fault.

{bf D} An independent fault in the decider (acceptance test, majority voter, comparator, adjudicator).

{bf RV##} (where each \# is an integer between 1 and 4) The input for a single computation activates a related fault between two versions. A related fault is one that occurs in two different versions causing both to produce the same erroneous result.

{bf RALL} A related fault affects all versions as well as the decider, caused by imperfect specifications.

{bf H#}(where \# is an integer between 1 and 4) A hardware fault affects the task computation.

Furthermore, let P_X is the probability that event X occurs, and $Q_{X} = 1 - P_{X}$, then the probability that an unacceptable result is produced during a single task iteration is given by

R(DRB) = P_{RV}+ Q_{RV} P_{D}+
Q_{RV} P_{RALL} Q_{D}+
Q_{RV} Q_{RALL} Q_{D} P_{H}^2 +
P_{V}^2 Q_{RV} Q_{RALL} Q_{D} (1-P_{H}^2)

3.3 Modeling Tools

The usage of software tools is a must in the modeling and analysis phase. We consider SHARPE [11] and UltraSAN [12] as two leading tools in this arena.

SHARPE (Symbolic Hierarchical Automated Reliability and Performance Evaluator) is a very general purpose performance and reliability modeling "toolchest" which allows the flexibility of choosing from various model types and hierarchically combine them, as per the demands of a particular problem. The model types currently supported by SHARPE include reliability block diagrams, reliability graphs, fault trees, Markov and semi-Markov chains, Markov and semi-Markov reward models, product-form queueing networks, generalized stochastic Petri nets, and series-parallel directed acyclic graphs. The tool enables computation of steady state as well as transient measures. The presence of many model types and the flexibility of model composition make SHARPE useful as a tool for analyzing real-world problems, and as a workbench for experimenting with modeling techniques, especially the use of exact and approximate system or model decomposition.

UltraSan (Ultra Stochastic Activity Networks) is a software tool for model-based performance, dependability and performability evaluation of computer, communication and other systems. The tool provides high-level modeling constructs in the form of stochastic activity networks (SANs), and offers hierarchical modeling by means of composed models. To specify performance and dependability measures for these models, reward variables are used. Given the SAN, composed model and reward variables, the tool either generates an executable discrete-event simulation or an underlying stochastic process, which then is solved by analytic methods. This tool provides six analytic solvers and three discrete-event simulators, one based on importance sampling. Furthermore, the report generator facilitates the generation of graphs and tables from the obtained performance results.

4. Phase 2: Design and Implementation Phase

In the Design and Implementation Phase, reliability of the software system is being achieved by software engineering techniques. The key topic of this phase is to provide fault avoidance and fault tolerance. Fault avoidance is the subject of many software engineering techniques and is beyond the scope of this paper. Fault tolerance, however, is the focus of our discussion. We examine fault tolerance techniques used in single version as well as multiple version environments.

4.1 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, atomicity of actions, decision verification, and exception handling. The most successful approach is primarily accomplished by reusable software fault tolerance routines [13]. 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 that are changing this tradition for providing fault tolerance. 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, there should be a mechanism to *detect and restart* a failed processes at the minimum. The next higher level is to checkpoint and recover 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 reactive recovery procedures, there is a complementary pro-active 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.

Implementing the above software fault tolerance and rejuvenation tasks individually in each application requires expertise in reliability and should be accomplished in a systematic fashion. We have developed a middleware platform containing a set of reusable software components ({\tt watchd}, {\tt libft}, {\tt REPL}, {\tt libckp}, and {\tt addrejuv}) to perform those tasks have been developed in [13]; see Figure 3.

Figure 3 Software Fault Tolerance Platform and Components

The hardware platform for using those reusable software components 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, log messages, watch, detect, rollback, restart, and recover from failures and rejuvenate to avoid failures. They are described as follows:

\bu {Watchd}

{\tt Watchd} is a watchdog daemon process that runs on a single machine or on a network of machines to detect application process failures and machine crashes. It determines whether a process is hung or not by either polling the application or checking a heartbeat message periodically sent from the application process to {\tt watchd}. When {\tt watchd} detects that an application process crashed or hung, 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. If {\tt libft} is also used, {\tt watchd} sets the restarted application to process all the logged messages from the log file generated by {\tt libft}.

{\tt Watchd} also facilitates restoring the saved values and re-executing the logged events. It also provides facilities for rejuvenation, remote execution, error reporting, remote copy, distributed election, and status report production. Several commands are also provided for operating, administrating and maintaining a network using {\tt watchd} daemons.

\bu {Libft}

{\tt Libft} is a user-level library of C functions 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 (e.g. {\tt critical()}) to specify critical volatile data 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.

{\tt Libft} also provides reliable read and write operations to automatically log messages. The logged data is then duplicated and logged by the {\tt watchd} daemon on a backup machine. The replication of logged data is necessary for a process to recover from a primary machine failure.

\bu {REPL}

{\tt REPL} is a file replication mechanism running on a pair of machines for on-line replication of critical files of an application. The mechanism uses dynamic-shared libraries to intercept file system calls. When a user program issues a file update, the shared library intercepts the request, performs the update locally, and passes the update message to a remote {\tt REPL} server. Upon receiving the message, the remote {\tt REPL} server replays the message and performs the file update. The critical files are specified through an environment variable. {\tt REPL} is built on top of standard file systems, and so its use requires no change to the underlying operating system. Speed, robustness and replication transparency are the primary design goals of the

{\tt REPL} replication mechanism.

\bu {Libckp}

{\tt 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 without requiring any modification to the source code. The checkpointed program state includes program counter, stack pointer, program stack, open file descriptors, the global/static variables and dynamically allocated memory of the program and the libraries linked with the program. {\tt Libckp} has two unique features. First, the library allows a user to include files as part of the process state that is checkpointed and recovered. More specifically, when a process rolls back, all the modifications it has made to the files since the last checkpoint are undone so that the states of the files are consistent with the checkpointed state. Other checkpointing libraries either do not support the rollback of user files or only provide the capability to a limited extent. The second unique feature of {\tt libckp} is that it also provides a non-transparent mode for flexible execution control: it provides application-initiated checkpoint and rollback facilities within a program. The rollback function rolls back the process to a a location in the program where the previous checkpoint was made. This facilitates restoration of global/static variables, dynamically allocated memory, and user files.
\bu {Addrejuv}

{\tt Addrejuv} is an added feature of {\em 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 [14]. When the {\tt addrejuv} feature is used, {\tt 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 three steps to stop the process. First, a signal or a command, specified as first argument to the {\tt addrejuv} feature, is sent to the process to kill it. Then, fifteen seconds later, a second signal or command, specified as second argument to the *addrejuv* feature, is sent to the process. Finally, fifteen seconds later, a SIGKILL signal is sent to the process to make sure that the process is really terminated. The fifteen seconds interval between the two signals allows the process to clean up its state before being terminated; the default value of fifteen seconds can be changed by the application. 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.

4.2 Multiple Version Software Fault Tolerance

Multiple, redundant computing channels (or "lanes") have been widely used in sets of *N = 2*, *3*, or *4* to build fault-tolerant hardware systems. To make a simplex software unit fault-tolerant, the corresponding solution is to add one, two, or more simplex units to form a set of *N ? 2* units. The redundant units are intended to compensate for, or mask a failed software unit when they are not affected by software faults that cause similar errors at cross-check points. The critical difference between multiple-channel hardware systems and fault-tolerant software units is that the simple replication of one design that is effective against random physical faults in hardware is not sufficient for software fault tolerance. Copying software will also copy the dormant software faults; therefore each simplex unit in the fault-tolerant set of *N* units needs to be built separately and independently of the other members of the set. This is the concept of software *design diversity* [15].

A set of *N ? 2* diverse simplex units alone is not fault-tolerant; the simplex units need an *execution environment* (EE) for fault-tolerant operation. Each simplex unit also needs fault tolerance features that allows it to serve as a *member* of the fault-tolerant software unit with support of the EE. The simplex units and the EE have to meet three requirements: (1) the EE must provide the support functions to execute the *N ? 2* member units in a fault-tolerant manner; (2) the specifications of the individual member units must define the fault tolerance features that they need for fault-tolerant operation supported by the EE; (3) the best effort must be made to minimize the probability of an undetected or unrecoverable failure of the fault-tolerant software unit that would be due to a single cause.

The evolution of techniques for building fault-tolerant software out of simplex units has taken two directions. The two basic models of fault-tolerant software units are *N-version software* (NVS), shown in Figure 4 and *recovery blocks* (RB) shown in Figure 5. The common property of both models 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 to be produced by the fault-tolerant unit. The NVS approach employs a generic *decision algorithm* that is provided by the 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 by necessity be *specific* for every distinct service, i.e., it is custom-designed for a given application, and is a member of the RB fault-tolerant software unit, but not a part of the EE.

Figure 4 The N-version software (NVS) model with n = 3

Figure 5 The recovery block (RB) model

N = 2 is the special case of *fail-safe* software units with two versions in NVS, and one alternate with one acceptance test in RB. They can detect disagreements between the versions, or between the alternate and the acceptance test, but cannot determine a consensus in NVS, or provide a backup alternate in RB. Either a *safe shutdown* is executed, or a *supplementary recovery process* must be invoked in case of a disagreement.

Both RB and NVS have evolved procedures for error recovery. In RB, backward recovery is achieved in a hierarchical manner through a {\it nesting} of RBs, supported by a {\it recursive cache}, or *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.

It is evident that the RB and NVS models converge if the acceptance test is done by NVS technique, i.e., when the acceptance test is specified to be one or more independent computations of the same outputs, followed by a choice of a consensus result. It must be noted that the individual versions of NVS usually contain error detection and exception handling (similar to an acceptance test), and that the NVP decision algorithm takes the known failures of member versions into account.

5. Phase 3: Testing and Measurement Phase

In this phase the reliability of the software system should be evaluated and verified, and testing and measurement techniques are available to achieve this goal. Testing techniques are for fault removal purpose, and reliability assessment techniques are for fault prediction purpose. We discuss each of them in the following sections.

5.1 Software 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. It is this structural coverage and its measurement which we believe is of value in reliability estimation. We describe two coverage testing methods, mutation testing and data and control flow testing. Subsequently we discuss the use of these methods in reliability estimation.

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. A decision refers to a simple condition. Thus, for example, the C language statement *if (a<b || p>q)...* consists of two simple conditions, $a<b$ and $p>q$, and one compound condition. We say that a decision is {\em covered} if during some execution it evaluates to true and in the same or another execution it evaluates to false. In the above example, the two simple conditions must evaluate to true and false during some execution for the decision coverage criterion to be satisfied.

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. A path from S_1 to S_2 is said to be *definition free* if no statement along this path, other than S_1 and S_2, defines *x*. Such a path is considered feasible if there exists at least one d ? D such that when *P* is executed on *d* the path is traversed.

All statements in *P* that can possibly be executed immediately after the execution of some statement *S*, are known as *successors* of *S*. We say that a c-use or a p-use is *covered* if the execution of *P* on some set of test cases causes at least one definition free path to be executed from the defining statement to the statement in which the use occurs and to each of its successors.

The above coverage measures 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. Using ATAC, two real-world applications were made available to show the relationship between testing and reliability. The first application is a 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 [16], using an N-version programming design paradigm. 12 versions of the autopilot program were produced and accepted at the end of the project. Figure 6 shows the progress of software testing from unit testing (1 complete 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 6 Relationship between Coverage Improvement and Fault Detection during Testing Phases

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 [17]. From this figure, again, we can see a clear relationship between high statement coverage in unit testing and low system test faults.

Figure 7 Relationship of Unit Coverage Testing to System Test Faults for One System

5.2 Software Reliability Measurement and Tool

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 8.

Figure 8 Software Reliabilty Measurement Procedure Overview

First, customer usage is quantified by developing an operational profile. Second, quality is 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, etc.) to maximize customer satisfaction. We then advocate the employment of operational profile and quality objectives to manage resources and to guide design, implementation, and testing of software. Moreover, we track reliability during testing to determine product release, using appropriate software reliability measurement tools. This activity may be repeated until a certain reliability
level has been achieved. We also analyze reliability in the field to validate the reliability engineering effort and to introduce product and process improvements.

It can be seen from Figure 8 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*.

A reliability objective is the specification of the reliability goal of a product from the viewpoint of the customer. If a reliability objective has been specified by the customer, that reliability objective should be used. Otherwise, you can select a reliability measure which is most intuitive and easily understood, and then determine the customer's "tolerance threshold" for system failures in terms of this reliability measure. For example, customer A might be mostly concerned with the total number of field failures product X may produce. Therefore, the reliability objective could be specified as, say, "product X should not produce more than 10 failures in its first 50 months of operation by customer A."

Operational profile is a set of disjoint alternatives of system operation and their associated probabilities of occurrence. 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.

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. Several interdependent estimates make equivalent statements about a product's reliability. They typically include the product's failure intensity (failure rate, i.e., the number of failures per unit time) as a function of test time t, the number of failures expected up to test time t, and the mean time to failure (MTTF) at test time t. 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 (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 already performed. This can be, for example, the product's reliability in the field, if the system testing has already been completed, or the predicted reliability at the end of testing, if the system testing has not yet been completed.

Despite the existence of more than 40 models, the problem of model selection and application is manageable. Experience has shown that it is sufficient to consider only a dozen models, including Jelinski-Moranda Model, Generalized Poisson Model, Goel-Okumoto Model, Musa Basic Model, Musa-Okumoto Model, Schneidewind Model, Non-Homogeneous Poisson Process Model, Delayed S-Shape Model, and Littlewood-Verrall Bayesian Model, etc.

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 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. When the operational profile is not fully developed, 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 trying to "break" the software by searching through the input space for difficult execution conditions, while users during operation only execute the software at a normal 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 software reliability measurement process for process improvement and better parameter tuning. For example, the model validity could be established, the growth of reliability could be determined, and the test compression factor could be refined, etc. 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 [18], 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 9 shows the high-level architecture for CASRE.

Figure 9 High-Level Architecture for CASRE

CASRE is currently executed in a Windows environment. 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 interfailure times) or Failure-Count models (for failure intensities). After one or more models have been executed, the predicted failure intensities or interfailure 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. Namely, 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 [19]. CASRE incorporates our findings that prediction accuracy may be increased by combining the results of several models in a linear fashion. Moreover, CASRE allows users to define their own combinations and record them as part of the tool's configuration. Weights for the components of the combination may be static or dynamic, and may be based on statistical techniques used to determine the applicability of a model to a set of failure data. Once combination models have been defined, the steps required to execute them are no different than executing a simple model. CASRE have 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 included in 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 planning, design, testing, and evaluation of software reliability. We integrate these techniques in a unified paradigm, consisting three software reliability engineering phases: (1) modeling and analysis, (2) design and implementation, (3) testing and measurement. We describe the reliability techniques associated with each of these three phases for fault management, fault avoidance and fault tolerance, as well as fault removal and fault prediction. We also discuss the software tools available in each phase, including SHARPE, UltraSAN for phase (1), watchd, libft, libckp, REPL, addrejuv for phase (2), and ATAC, CASRE for phase (3). We examine CASRE in detail for its capability to apply multiple software reliability models and to choose the most appropriate model for project-specific environments.

6. Summary of Due Dates

The key dates for conference abstract and paper submission, review, and final copy plus the dates for registration and lodging payments are summarized in Table 1.

7. Summary of Style Specifications

The editorial style requirements for 1997 IEEE Aerospace Conference Papers are summarized in Table 2.

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

[8] E.J. Henley and H. Kumamoto, Probabilistic Risk Assessment, IEEE Press, New York, 1982.

[9] K.H. Kim and H.O. Welch, "Distributed Execution of Recovery Blocks: An Approach for Uniform Treatment of Hardware and Software Faults in Real-Time Applications," IEEE Transactions on Computers, 38:5, 626-636, May 1989.

[10] J.-C. Laprie, J. Arlat, C. Beounes, and K. Kanoun, "Definitiona and Analysis of Hardware- and Software-Fault-Tolerant Architectures," IEEE Computer, 23:7, 39-51, July 1990.

[11] R.A.Sahner, K.S. Trivedi, and A. Puliafito, Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using the SHARPE Software Package, Kluwer Academic Publishers, Boston, MA, 1996.

[12] J. Couvillion, R. Freire, R. Johnson, W.D. Obal, M.A. Qureshi, M. Rai, W.H. Sanders, and J.E. Tvedt, "Performability Modeling with UltraSan," IEEE Software, 8:5, 69-80, Sept. 1991.

[13] 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.

[14] 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.

[15] A. Avizienis, "The Methodology of N-Version Programming," Chapter 2 of Software Fault Tolerance, M. Lyu (ed.), Wiley, 23-46, 1995.

[16] M.R. Lyu and Y. He, "Improving the N-Version Programming Process Through the Evolution of a Design Paradigm," in IEEE Transactions on Reliability, 42:2, 179 - 189, June 1993.

[17] 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.

[18] 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.

[19] M.R. Lyu and A. Nikora, "Using Software Reliability Models More Effectively," IEEE Software, 43-52, July 1992.

***Michael R. Lyu** is currently a Member of the Technical Staff at Bell Labs Research, Lucent Technologies. He worked at the Jet Propulsion Laboratory as a Member of the Technical Staff from 1988 to 1990. From 1990 to 1992 he was with the Electrical and Computer Engineering Department at the University of Iowa as an Assistant Professor. From 1992 to 1995, he was a Member of the Technical Staff in the Applied Research Area of the Bell Communications Research(Bellcore). Dr. Lyu's research* *interests include software reliability engineering, software process and metrics, distributed systems, and fault-tolerant computing. He has published over 50 refereed journal and conference papers in these areas. He initiated the first International Symposium on Software Reliability Engineering (ISSRE) in 1990. He was the program chair for ISSRE'96, and has served in program committees for many conferences. He is the editor for two book volumes: Software Fault Tolerance, published by Wiley in 1995 and the Handbook of Software Reliability Engineering, published by IEEE and McGraw-Hill in 1996. He is an associated editor of IEEE Transactions on Reliability and an editor for IEEE Transactions on Knowledge and Data Engineering.*