Reliability-oriented software engineering: design, testing and evaluation techniques

M.R.Lyu

Indexing terms: Reliability, Fault tolerance, Fault removal

Abstract: Software reliability engineering involves techniques for the design, testing and evaluation of software systems, focusing on reliability attributes. Design for reliability is achieved by fault-tolerance techniques that keep the system working in the presence of software faults. Testing for reliability is achieved by fault-removal techniques that detect and correct software faults before the system is deployed. Evaluation for reliability is achieved by fault-prediction techniques that model and measure the reliability of the system during its operation. This paper presents the best current practices in software reliability engineering for design, testing and evaluation purposes. There are descriptions of how fault-tolerant components are designed and applied to software systems, how software testing schemes are performed to show improvement of software reliability, and how reliability quantities are obtained for software systems. The tools associated with these techniques are also examined, and some application results are described.

1 Introduction and overview

Software has become the bottleneck of system development, and its delays and cost overruns have often put modern, complex projects in jeopardy. Moreover, computer software has already become the major source of reported outages in many systems [1]. Fig. 1 shows the causes of the total outage incidents of US switching systems in 1992 [2], 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.

As a result, software industries have seen a major share of project development costs associated with the design, implementation and assurance of reliable software, and people have recognised 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 related disciplines.

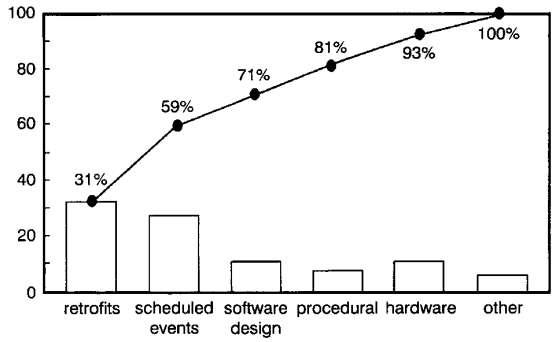

Fig. 1 *Switching system outage causal classification*
Total percentage related to software = 81

Software reliability engineering [3] is centred 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 [4]. It is one of the attributes of software quality, a multi-dimensional property including other factors such as functionality, usability, performance, serviceability, capability, installability, maintainability and documentation [5]. Software reliability, on the other hand, is generally accepted as the key factor in software quality, as it is aimed at quantifying and predicting software failures, the unwanted events that can make a powerful system inoperative or even deadly. Thus reliability is an essential ingredient for customer satisfaction for most commercial companies and government organisations. 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 from the customer's viewpoint. ... The supplier of software products should collect and act on quantitative measures of the quality of these software products.' (See Section 6.4.1 of [6]).

In this paper, we discuss software reliability engineering techniques in three categories: design for software reliability; testing for software reliability; and 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 a fault-tolerance capability. The available techniques we emphasise 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 is to

IEE Proceedings online no. 19982439
Paper received 21st July 1998

The author is with the Computer Science and Engineering Department, The Chinese University of Hong Kong, Shatin, Hong Kong

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 modelling techniques. The key 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 Sections.

2 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.

2.1 Single-version software fault tolerance

Software fault tolerance in a 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. In the current marketplace, standard commercial hardware and operating systems are becoming more reliable, distributed and inexpensive. They are now off-the-shelf, commodity items with open systems and evolving standards and interfaces. Furthermore, the proportion of failures resulting from faults in the application software is increasing owing to the increased size and complexity of software.

To implement application-level software fault tolerance, at the baseline level we need a mechanism to detect and restart failed processes at the minimum. In addition, we can perform checkpoint and recovery for the internal state of a process when it fails. Furthermore, logging and replaying messages can be used. It can happen that some part of the environment will change during recovery and replay in such 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 above reactive recovery procedures, 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, reinitialising the in-memory volatile data structures and logging administrative records.

Fig. 2 shows a middleware platform, software implemented fault tolerance (SwIFT), which includes a set of reusable software components (watchd, libft, libckp, REPL 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 proactively.

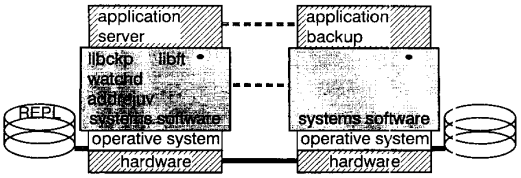

Fig. 2 *SwIFT platform and components*

Watchd is a watchdog daemon process, the purpose of which is to detect application process failures and machine crashes. It runs on a single machine or on a network of machines and 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 has crashed or failed, it recovers that application at an initial internal state or at the last checkpointed state. It is recovered either on the primary node, or on the backup node.

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.

Libckp is a user-transparent checkpointing library. It can be linked with a user's program to save the program state periodically 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.

REPL is a file replication mechanism for on-line replication of the 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.

Addrejuv is an added feature of watchd that performs software rejuvenation. 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 termi-

nated, watchd takes a recovery action to re-spawn the process in the same manner as it does when it detects a failure.

2.2 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 Fig. 3, and recovery blocks (RBs), shown in Fig. 4.

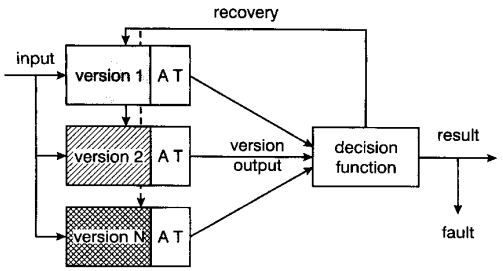

Fig. 3 *N-version software model*

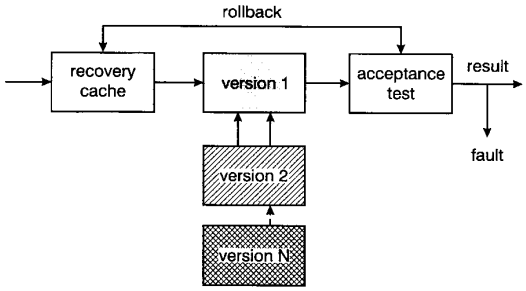

Fig. 4 *Recovery block model*

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 used 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 uses a generic decision algorithm that is provided by the execution environment 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 custom-designed for a given application and is a member of the RB fault-tolerant software unit.

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. In NVS, forward recovery is achieved by the use of the community error recovery algorithm, which is supported by the specification of recovery points and by a generic decision algorithm. 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 Fig. 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

(*a*) to attain suitable isolation and independence (with respect to software faults) of the *N* concurrent version development efforts

(*b*) to encourage potential diversity among the *N* versions of an *N*-version software unit

(*c*) to elaborate efficient error detection and recovery mechanisms.

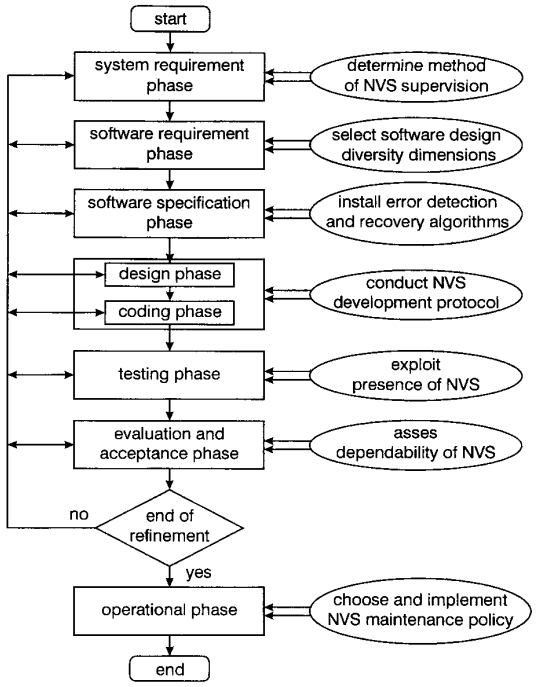

Fig. 5 *Design paradigm for NVP*

3 Testing for software reliability

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

3.1 Software coverage testing scheme and tool

There are many ways to test software. The terms junctional, regression, integration, product, unit, coverage and user-oriented are only a few of the characterisations we encounter. These terms are derived from the method of software testing or the development phase during which the software is tested. In particular, white-box, or coverage, testing uses the structure of the software to measure the quality of testing. This structural coverage measurement is closely related to reliability estimation. Coverage 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 is 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 defined as a c-use. If the use appears inside a predicate, then the pair is defined as a p-use.

Coverage measures from the above testing criteria are obtainable from the automatic test analysis for C (ATAC) tool. ATAC 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. computerised) aeroplane 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 Fig. 5. Twelve 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 Fig. 6.

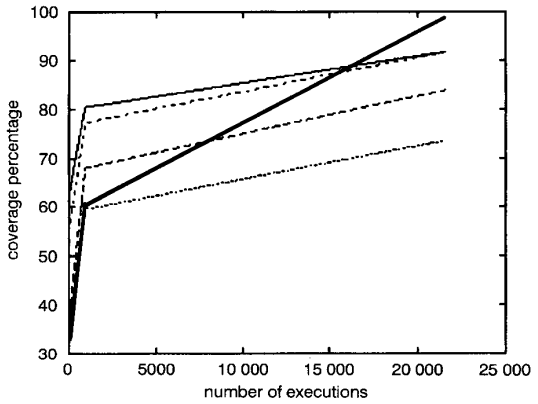

Fig. 6 *Relationship between coverage improvement and fault detection*
Block coverage
Decision coverage
c-use coverage
p-use coverage
Known faults detected

Fig. 6 shows the progress of software testing from unit testing (one complete flight simulation test case), integration testing (960 test cases), to acceptance testing (21 600 test cases). The broken lines depict the accumulation of test coverage, and 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 Fig. 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.

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

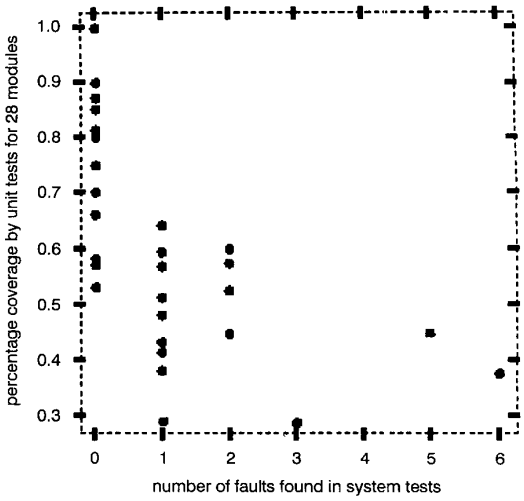

Fig. 7 *Relationship of unit coverage testing to system test faults for one system*

3.2 Software fault insertion testing

Another specific testing scheme is software fault insertion testing (SFIT). The main objective of SFIT is to test a system's fault-tolerance capability through injecting faults into the system and then observing whether the system can detect these faults and recover from various scenarios. SFIT is recommended for system testing or acceptance testing during the testing life cycle. In this way, the system's overall reaction to faults can be observed and analysed. 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.

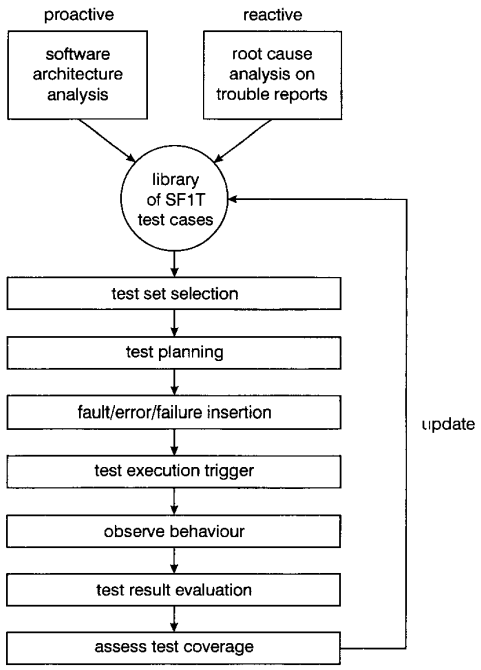

Fig. 8 *Software fault insertion testing methodology*

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

Step 1: Software architecture analysis: before conducting SFIT, a sufficient knowledge of the software (including functions of some key software components, such as the error-recovery software subsystem) and its architecture is required. This analysis is proactive and includes three key parts: application software analysis, fault manager analysis and interface analysis.

Step 2: Root cause analysis: internal testing results and external field problems can often 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 organisation 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 to be tested.

Step 3: Test set selection: during the test set selection, the following two aspects need to be identified:

(*a*) properties and predicates to be checked to assess the expected behaviour of the system in the presence of the injected faults

(*b*) observations to be made to verify the assertion of the corresponding actual system behaviour.

Step 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. Faults can be injected either in software code or in system state. 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.

Step 5: Fault insertion: with all the test cases available, this step involves the actual insertion of faults in the code or in the state. The location of faults should be identified during this step.

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.

Step 7: Observe behaviour: this step observes the system reaction to the inserted faults within a specified time frame.

Step 8: Test result evaluation: the test result can reveal the effectiveness of the test cases as well as the 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.

Step 9: Assess test coverage: although complete testing coverage with SFIT is not economically possible, a notion of test coverage adequacy is essential to confidence in the fault tolerance of a system.

Step 10: Update SFIT test case library: a library of common and generic faults, 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. Subsequently, an adequacy criterion for fault-insertion testing can be gradually established.

4 Evaluation for software reliability

Evaluation for reliability is focused on the modelling and analysis techniques for fault-prediction purposes. We discuss a systematic software reliability measurement procedure and a software reliability estimation tool.

4.1 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 purposes, as described in Fig. 9.

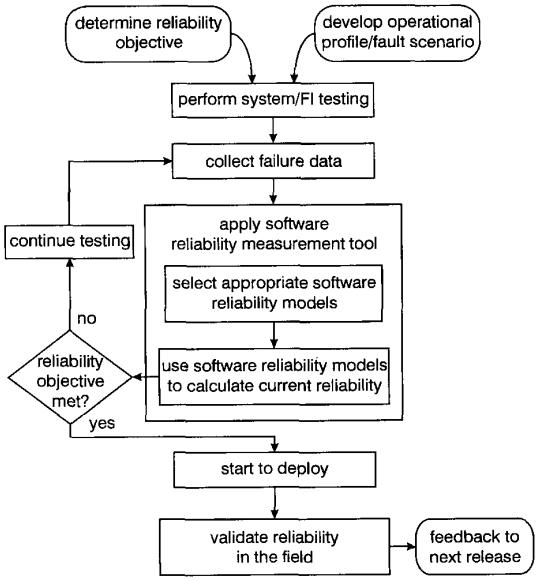

Fig. 9 *Software reliability measurement procedure overview*

Fig. 9 describes four major components in this software reliability measurement process, namely, reliability objective; operational profile; reliability modelling and measurement; and reliability validation.

According to this framework, customer-perceived software quality is first quantitatively defined by examining failures and failure severity, by determining a reliability objective, and by specifying balance among key quality objectives (e.g. reliability, delivery date, cost).

Secondly, customer usage is quantified by developing an operational profile. The operational profile is a set of disjoint alternatives of system operation and their associated probabilities of occurrence (see chapter 5 in [3]). 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, the 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 can be repeated until a certain reliability level has been achieved. Finally, reliability can be analysed in the field to validate the reliability engineering effort and to provide feedback for product and process improvements.

Reliability modelling is an essential element of the reliability estimation process, which determines whether a product meets its reliability objective and is ready for release. A reliability model calculates, 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 time. These reliability estimates can provide the following information, useful for product quality management: the reliability of the product at the end of system testing; the amount of (additional) test time required to reach the product's reliability objective; the reliability growth as a result of testing; and the predicted reliability beyond the system testing already performed.

4.2 Software reliability measurement tool

There are as many as 40 software reliability models proposed in the literature. Despite the existence of a large quantity of (and variation in) these models, the problem of model selection and application is manageable. Using the statistical methods provided in [3] (chapter 4), 'best' estimates of reliability can be obtained during testing. These estimates are then used to project the reliability during field operation to determine whether the reliability objective has been met. This procedure is an iterative process, as more testing will be needed if the objective is not met.

As 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 modelling tool, called the computer-aided software reliability estimation (CASRE) system [15], for an automatic and systematic approach to estimating software reliability. CASRE is implemented as a software reliability modelling tool that addresses the ease-of-use issue as well as other issues. Fig. 10 shows the high-level architecture for CASRE.

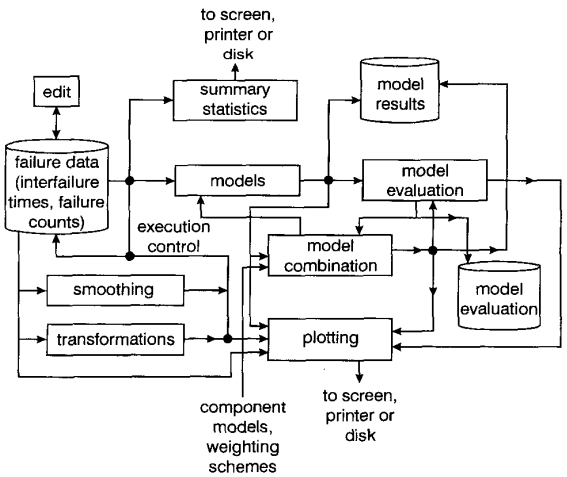

Fig. 10 *High-level architecture for CASRE*

CASRE is designed for the Windows environment. A Web-based version is also available [16]. The command interface is menu driven; users are guided through the selection of a set of failure data and executing a model by selectively enabling pull-down menu options. Modelling results are also presented in a graphical manner. Users can select multiple models from two categories, depending on the 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 can 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, the predictive quality of which is better than that of the individual models themselves [17]. CASRE has been used by major corporations including AT&T, Lucent, Microsoft, NASA, IBM, Motorola, Nortel etc. (Details of this tool can be found at the website http://www.cse.cuhk.edu.hk/~lyu/book/reliability).

5 Conclusions

Developing reliable software systems is a formidable task that involves the best of our knowledge of 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.

6 Acknowledgments

I would like to thank Prof. Andy Tyrrell of the University of York for his encouragement of this publication.

This work is supported by a CUHK Direct Grant (project code 2050182).

7 References

1 GRAY, J.: 'A census of Tandem system availability between 1985 and 1990', *IEEE Trans. Reliab.*, October 1990, **39**, (4), pp. 09–418
2 National Reliability Council (NRC): 'Switch focus team report'. June 1993
3 LYU, M.R. (Ed.): 'Handbook of software reliability engineering' (McGraw-Hill and IEEE Computer Society Press, New York, 1996, 1st edn.)
4 Institute of Electrical & Electronics Engineers: 'ANSI/IEEE standard glossary of software engineering terminology'. IEEE Std. 729–1991, 1991
5 GRADY, R.B.: 'Practical software metrics for project management and process improvement' (Prentice-Hall, Englewood Cliffs, New Jersey, 1992)
6 International Standard Organisation: '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
7 HUANG, Y., KINTALA, C.M.R., BERNSTEIN, L., and WANG, Y.M.: 'Components for software fault tolerance and rejuvenation', *AT&T Tech. J.*, March/April 1996, pp. 29–37
8 HUANG, Y., KINTALA, C.M.R., KOLETTIS, N., and FULTON, N.D.: 'Software rejuvenation: analysis, module and applications',25th international symposium on *Fault-tolerant computing* FTCS-25, Pasadena, California, June 1995 pp. 381–390
9 AVIZIENIS, A.: 'The methodology of n-version programming' *in* LYU, M.R. (Ed.): 'Software fault tolerance' (Wiley, 1995), Chap. 2, pp. 23–46

10 LYU, M.R.: 'A design paradigm for multi-version software'. PhD Dissertation, Computer Science Department, UCLA, May 1988
11 LYU, M.R., HORGAN, J.R., and LONDON, S.: 'A coverage analysis tool for the effectiveness of software testing', *IEEE Trans. Reliab.*, December 1994, **43**, (4), pp. 527–535
12 LYU, M.R., and HE, Y.: 'Improving the n-version programming process through the evolution of a design paradigm', *IEEE Trans. Reliab.*, June 1993, **42**, (2), pp. 179–189
13 DALAL, S.R., HORGAN, J.R., and KETTENRING, J.R.: 'Reliable software and communication: software quality, reliability, and safety'. 15th international conference on *Software engineering*, Baltimore, MD, May 1993
14 LAI, M.Y., and WANG, S.Y.: 'Software fault insertion testing for fault tolerance' *in* LYU, M.R. (Ed.): 'Software fault tolerance' (Wiley, 1995), Chap. 13, pp. 315–333
15 LYU, M.R., and NIKORA, A.: 'CASRE - a computer-aided software reliability estimation tool'. 1992 *Computer-aided software engineering* workshop, Montreal, Canada, July 1992, pp. 264–275
16 LYU, M.R., and SCHOENWAELDER, J.: 'Web-CASRE: a Web-based tool for software reliability measurement'. International symposium on *Software reliability engineering*, ISSRE'98, Paderborn, Germany, November 1998
17 LYU, M.R., and NIKORA, A.: 'Using software reliability models more effectively', *IEEE Softw.*, July 1992, pp. 43–52