*Proceedings of the 8th International Symposium on Software Reliability Engineering*, Albuquerque, NM, USA, Nov. 1997, pp. 192-202.

# An Approach to Measuring and Assessing Dependability for Critical Software Systems

Dong Tang and Herbert Hecht
SoHaR Incorporated
Beverly Hills, California, USA

## Abstract[1]

*Traditional software testing methods combined with probabilistic models cannot measure and assess dependability for software that requires very high reliability (failure rate < $10^{-6}$/hour) and availability (>0.999999). This paper proposes a novel approach, drawing on findings and methods that have been described individually but have never been combined, applied in the late testing phase or early operational phase, to quantify dependability for a category of critical software with such high requirements. The concepts that are integrated are: operational profile, rare conditions, importance sampling, stress testing, and measurement-based dependability evaluation. In the approach, importance sampling is applied on the operational profile to guide the testing of critical operations of the software, thereby accelerating the occurrence of rare conditions which have been shown to be a leading cause of failure in critical systems. The failure rates measured in the testing are then transformed to those that would occur in the normal operation by the likelihood ratio function of the importance sampling theory, and finally dependability for the tested software system is evaluated by using measurement-based dependability modeling techniques. When the acceleration factor is large (over 100), which is typical for a category of software of interest, it is possible to quantify a very high reliability or availability in a reasonable test duration. Some feasible methods to implement the approach are discussed based on real data.*

## 1. Introduction

In all critical applications where software plays a decision-making or control role, the quantification of dependability is recognized as a vital issue that currently is not satisfactorily resolved. That traditional software testing methods combined with reliability models are not suitable for applications with very high reliability requirements (failure rate < $10^{-6}$/hour) has been stated by several researchers in this field [6, 28]. Reliability models based on software fault tolerance by design diversity [4] face a challenge to their underlying independence assumption [14, 26]. The reliability improvement achievable by these methods is therefore not as high as expected; in addition, the design diversity approach is very expensive. In other words, none of the current testing and redundancy modeling techniques provides a satisfactory solution to reliability estimation for critical software.

We propose an approach that can partially address the above issue for a category of software systems based on the following elements:

• *Operational profile*, a representative probability distribution of actual operations that the target software will be performing in the field which has been used in recent years to guide testing and reliability assessment by major companies [33, 35].

• *Rare conditions* (or rare events), an important cause of failures that is clearly seen in data from the NASA space shuttle software [18, 19] and from several well known critical software failure events such as the Ariane rocket failure [43] and the Seattle air traffic control system failure [42]. Rare conditions are typically generated by a combination of erroneous or unexpected input data and rarely executed faulty code.

• *Importance sampling*, a statistical method to reduce sampling size while keeping estimates obtained from the sample unbiased at a high level of confidence. The method has been used to evaluate dependability measures from rare events in Monte Carlo simulations to solve large Markovian models [16] and to inject faults to microprocessors [9].

• *Stress testing*, accelerated testing of some operations of the software such that their occurrence probabilities are much greater than in the operational profile. This idea has actually been applied by researchers in developing benchmarks to measure system robustness [37] and in testing fault tolerance by fault injections [5].

• *Measurement-based evaluation* of software dependability from test and field data, an extension of measurement-based dependability analysis methodologies [21], has been shown feasible by authors' recent work [40, 42]. This technique can account for the structure and fault

[1]This research was supported by the U.S. National Science Foundation under Grant DMI-9661512.

tolerance of a program, e. g., failures in one software task are (in many cases) compensated for by running a backup task later.

The proposed approach applies importance sampling on the operational profile to guide testing critical components in the software to accelerate the occurrence of rare conditions, then transforms failure rates measured in the testing to those that would occur in the normal operation by the likelihood ratio function of the importance sampling theory, and finally evaluates dependability for the tested software system based on the transformed failure rates (notice these rare events dominate the dependability of the software) using measurement-based dependability modeling techniques. When the acceleration factor (likelihood ratio) is large (over 100), which is likely to be the case for a category of software where the occurrence probability of rare conditions in the normal operation is much lower than in the importance sampling testing (an example of such software will be discussed with Table 1), it is possible to quantify a very high reliability or availability in a reasonable test duration.

Applicable software systems are constrained by the following conditions: (1) operational profile for the software is identifiable; (2) critical operations constitute only a small part of the operational profile; and (3) failures of critical operations are the major threat to system dependability. In addition to these limitations, we would like to point out that by our experience, a dependability assessment that is accurate to the right order of magnitude (e.g., an availability of five 9's, regardless of 0.999992 or 0.999998) is more practical and reasonable than an assessment that claims to represent the exact actual dependability. This is because there are various factors that can bias results, such as errors in measurement, parameter estimation, model construction and model evaluation.

## 2. Related Research and Background

This section reviews related research and provides an overview of the importance sampling method.

### 2.1 Related Research

Research that addresses the quantification of software reliability has been active since the early 1970's [15]. Many reliability growth models have been proposed to characterize reliability growth for software under development. These models evaluate the reduction in failure frequency during successive test intervals to estimate the software reliability at the conclusion of the test. Some of the recommended models are the Schneidewind model, the generalized exponential model, the Musa/Okumoto Logarithmic Poisson model, and the Littlewood/Verrall model [3]. Applications of these models have all been demonstrated using real data from software with observed failure rates from $10^{-1}$ to $10^{-5}$ per hour [1, 23, 32].

It has been shown in [6] that the quantification of life-critical software reliability is infeasible by using reliability growth models along with traditional testing techniques. An illustrative example in the study is that it would take $10^{-8}$ to $10^{-10}$ hours (thousands of years) of testing to demonstrate a failure rate of $10^{-7}$ to $10^{-9}$ per hour assuming one copy of software would be tested and one failure would be observed. Even if 10 copies (which is likely) or 100 copies (which is unlikely) of the software are tested concurrently, it would still take hundreds (for 10 copies) or tens (for 100 copies) of years. The study also cited comments of other experts in the field on this issue, including the following:

"Clearly, the reliability growth techniques of §2 [a survey of the leading reliability growth models] are useless in the face of such ultra-high requirements. It is easy to see that, even in the unlikely event that the system had achieved such a reliability, we could not assure ourselves of that achievement in an acceptable time." [28]

Another limitation of reliability growth models is their lack of ability to model software structure. Reliability growth models treat the software as a black box and assess the reliability of the software as a whole. Critical software systems include fault tolerance mechanisms, such as error detection and handling, redundancy management, and backup tasks. As such, the dependability (reliability, availability, etc.) of the whole software system cannot be simply quantified by observed failures at the component (task) level. For example, a transient task failure may be covered by the fault tolerance provisions and may not affect any critical functions. This scenario has been verified by several studies [17, 31, 40] which showed that 80% to 95% of software failures in real-time fault tolerant systems are recoverable by provisions of redundant processes. In such a case, reliability growth models are not suitable for dependability quantification and structured dependability models need to be used.

Dependability models have been used to evaluate operational software based on failure data collected from commercial computer operating systems for over 10 years (e.g., [20, 30]). The methodology was recently extended to evaluate availability for air traffic control software systems in the late testing phase [40] and early operational phase with multiple sites [42]. In our experience, three model structures have been found most useful in measurement-based dependability evaluation: reliability block diagram, *k*-out-of-*n* model, and Markov chain. Both reliability diagrams and *k*-out-of-*n* models are combinatorial models and typically assume failure independence among modeled components. Markov chains are stochastic models which can

incorporate interactions among components and failure dependence.

However, the current practice of measurement-based evaluation for individual software systems (with the number of installations < 100) is still limited to failure rates of $10^{-2}$ to $10^{-5}$ per hour and an availability of three to five 9's (0.999 to 0.99999). For example, the newly developed FAA Voice Switching and Control System (VSCS) is being installed in 21 major U.S. air traffic control centers and the authors were tracking the system operational availability based on failure reports from installed sites during a period between 1995 and 1996. In this work, the system availability (dominated by software) was evaluated to have reached five 9's as of June 30, 1996. If no major failure occurs in the future, it will take 15 years of normal operation for the 21 sites to demonstrate an availability of the required seven 9's at the 80% confidence level, using the upper bound based evaluation method discussed in [40]. Therefore, in order to quantify availability for systems with such high requirements, it is necessary to explore accelerated testing and assessment methods.

A feasible acceleration approach is the stress testing of components or operations which can cause failures that are critical to the system. Stress testing techniques are sometimes used by software manufacturers to test rarely exercised components or operations, such as redundancy management software. A particular stress testing technique, fault injection, has also been used on software in laboratory studies in recent years [2, 5, 8, 24, 36, 37, 43]. But none of these stress testing techniques has been directly connected to reliability or availability quantification. Even if thorough testing has been performed and the software has actually reached a very high reliability, the lack of evaluation means prevents us from knowing what the software reliability is.

The key problem is that in the stress testing, the test cases are not very representative of the operational usage. The test cases used represent a biased operational profile and the failure rate measured is greater than the actual failure rate. If the degree of the amplification is known, it could be adjusted back to the actual value. Importance sampling is a statistical method that may offer a solution to this problem by reducing sampling size while keeping estimates obtained from the sample correct at a high level of confidence [22]. Specifically, it allows the analysis of rare events with a high degree of accuracy when Monte Carlo techniques are used. The method, summarized below, has been used in recent years to reduce the number of runs in Monte Carlo simulations for evaluating computer dependability [9, 16].

### 2.2 Overview of Importance Sampling

Assume that a random variable $X$ has probability density function (p.d.f.) $f(x)$ and that $Y=h(X)$ is a function of $X$. Our goal is to estimate the expected value of $Y$,

$$\theta = E[Y] = E[h(X)] = \int_{-\infty}^{+\infty} h(x)f(x)dx \quad \textbf{(1)}$$

through sampling. That is, we generate a sample $\{x_1, x_2, ..., x_n\}$ according to $f(x)$, therefore generating $\{y_1, y_2, ..., y_n\}$, and then calculate

$$\tilde{\theta} = \bar{Y} = \frac{1}{n}\sum_{i=1}^{n} y_i = \frac{1}{n}\sum_{i=1}^{n} h(x_i) \quad \textbf{(2)}$$

It may be very expensive to generate a statistically significant sample of $X$. For example, if $y_i = h(x_i) = 0$ for most generated $x_i$, we may need an extremely large size of sample to estimate $\theta$ with a high level of confidence. However, if we can make the rare $x_i$'s which are "important" for estimating $\theta$ be much more frequently selected in the sampling while keeping the estimate unbiased, the sample size would be greatly reduced. This is the basic idea of the importance sampling method.

In importance sampling, we change the p.d.f. of $X$ from $f(x)$ to $g(x)$ such that those $x$'s which are of importance in our parameter estimation have higher occurrence probabilities in $g(x)$. We use $X'$ to represent the variable which has p.d.f. $g(x)$. By Equation (1), we have

$$\theta = \int_{-\infty}^{+\infty} h(x)f(x)dx = \int_{-\infty}^{+\infty} h(x)\frac{f(x)}{g(x)}g(x)dx = \int_{-\infty}^{+\infty} h(x)\Lambda(x)g(x)dx \quad \textbf{(3)}$$

where

$$\Lambda(x) = \frac{f(x)}{g(x)} \quad \textbf{(4)}$$

is called *likelihood ratio*. Let $Y' = h(X)\Lambda(X)$, then Equation (3) becomes

$$\theta = \int_{-\infty}^{+\infty} y'g(x)dx = E[Y'] \quad \textbf{(5)}$$

Thus, instead of sampling from $f(x)$ to estimate the expected value of $Y$, the experiment is changed to sampling from $g(x')$ to estimate the expected value of $Y'$. That is, we generate a sample $\{x'_1, x'_2, ..., x'_n\}$ according to $g(x')$, therefore generating $\{y'_1, y'_2, ..., y'_n\}$, and then calculate

$$\tilde{\theta} = \bar{Y'} = \frac{1}{n}\sum_{i=1}^{n} y'_i = \frac{1}{n}\sum_{i=1}^{n} h(x'_i)\Lambda(x'_i) \quad \textbf{(6)}$$

## 3. Discussion of the Approach

Figure 1 shows the layout of the technical approach which integrates the five elements discussed in Section 1. The mapping between the operational profile and the stress testing profile is quantified by importance sampling. This quantification allows the estimation of failure rates, which depend on rare events, to be unbiased. The following subsections discuss feasible methods that can be used to implement the approach.

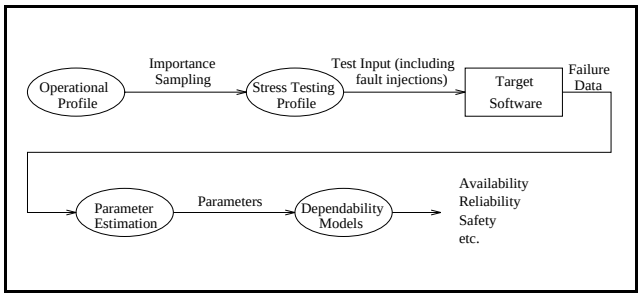


**Figure 1 Layout of the Approach**

### 3.1 Operational Profile

The term *profile*, as defined in [35], denotes "a set of disjoint alternatives called elements, each with the probability that it will occur." The operational profile is a quantitative characterization of how a software system will be used. It is desired to have a representative operational profile for evaluating software dependability by testing, although studies [11, 34] showed that reliability estimates are not very sensitive to operational profile errors. The procedure for developing operational profiles is discussed in detail in [32]. For use in the approach proposed in this paper, the operational profile should be classified by system modes and critical operations. Two key efforts are particularly important:

- Identification of critical system modes and their occurrence probabilities
- Identification of rarely executed critical operations and their occurrence probabilities

We consider two broad categories of systems: (1) continuously operating real-time systems, and (2) on-line protection systems. The operational profiles differ radically for the two categories: continuous input (workload may fluctuate) and intermittent input (rare events). The first category requires high availability and can tolerate component-level failures by redundancy provisions. Computer operating systems, telephone switching and air traffic control systems all fall into this category. The second category requires successful response to emergency demands and a failed response can result in the loss of life and property. The nuclear power safety system is a typical representative of this category.

In an air traffic control center, a failure of switchover from the failed, primary air to ground communication channel to a standby channel would cause a loss of communications between air traffic controllers and all planes in the affected area, creating a potential disaster. In a nuclear plant safety system, the significant challenges are events calling for shutdown and border line events in which shutdown is not required. These events occur at rates of one to five per year. Thus, critical system modes for these two categories include:

- failure recovery mode
- challenge processing mode

A recent study of failures in the U.S. Public Switched Telephone Network showed that in the software failure category, 70% (31 out of 44) occurred in the recovery mode [27]. This finding confirms the criticality of failure recovery mode. Typically, these critical modes are also rarely executed modes. But rarely executed operations are not limited in these modes. Based on past analyses of failure data from field software systems [18, 29, 38, 39, 42, 43], the following operations are often rarely executed and are likely to cause problems in the operational phase:

- redundancy management
- exception handling
- initialization and calibration
- processing of data in boundary conditions or out of expected ranges

After the identification of critical modes and rarely executed operations (for simplicity, *critical operations*), the next task is to estimate occurrence probabilities for these operations. Some of these operations are determined by the normal workload, such as initialization and calibration. Some of these operations are determined by the fault arrival rate, such as redundancy management. For example, if a process control program reinitializes itself every 100 hours and if each initialization takes five minutes, the occurrence probability for the initialization operation is 5-minutes/100-hours $\approx$ 0.00083. If faults that require channel switching arrive once every six months in a fault-tolerant system and if each channel switching operation takes 10 minutes, the occurrence probability for the redundancy management operation is 10-minutes/6-months $\approx$ 0.000038.

Sometimes it is very difficult to estimate individual occurrence probabilities for critical operations. However, it may not be very difficult to determine an upper bound that the total occurrence probability of the critical operations will not exceed. That is, we have

$$P_C < P_{upp} \quad \textbf{(7)}$$

where $P_C$ is the probability that the system is operating in the critical operations, which is unknown, and $P_{upp}$ is an upper bound of $P_C$, which can be estimated. We call $P_C$ the

*importance factor* and it is a key parameter to estimate in the proposed approach. For the NASA/JPL Deep Space Network discussed below, an estimated $P_{upp}$ is 0.01. The estimated $P_{upp}$ can be in place of $P_C$ to provide an upper bound of failure rate as discussed in Section 3.2.

*Examples*

The Ariane 5 rocket disaster occurring on June 4, 1996 was due to a data error in a computer and the unavailability of the software to handle the error. In a calibration function of the on-board Inertial Reference System (SRI), the horizontal velocity value was generated much higher than expected due to inconsistent software design. When the software tried to convert this 64-bit float to a 16-bit integer, the operation failed because the value was too large, thus sending off an exception. This caused the SRI to fail and to send faulty data to the On-Board Computer (OBC) which executes the flight program. The faulty data forced the OBC to change the angle of attack that led to separation of the boosters from the main stage, in turn triggering the self-destruct system of the launcher. [43]

The Voice Switching and Control System (VSCS) located at Seattle experienced a Type I failure (a problem that precludes the primary air traffic control system mission objective of controlling aircraft) on August 11, 1995 when all of the eight Air to Ground telephony switch shelves (A/G shelf) failed. The event was widely reported by the media. The problem diagnosis provided by the manufacturer identified the likely root cause as an undetected fault in a memory chip. The fault corrupted the length field (set to 0-length) of a message broadcast by an A/G shelf. All other A/G shelves (including primaries and standbys), upon receiving this invalid message, reset and cleared all application code from their processors simultaneously due to a general protection fault caused by the 0-length. [42]

Both of the above events were triggered by data being out of the expected ranges, although causes may differ (software and hardware problems). However, both events could be avoided if the corresponding data checking and protection code exist in the software. As suggested in [43], fault injection could have played a major role in avoiding such disasters. Fault injection is a method of the stress testing to be discussed later. In order to have properly directed fault injections, it is necessary to identify all critical system modes and operations that could lead to disasters in the development of operational profile.

Table 1 shows some characteristics of the failure data from the NASA/JPL Deep Space Network (DSN) [18], where FC denotes frequently executed code and RC denotes rarely executed code. In this program, the RC included redundancy management, exception handling, initialization and calibration routines. Failures in these functions were frequently much more critical to the value of the spacecraft data than failures affecting routine data transmissions. Also, it was estimated that over a period of several months the FC would be exercised at least 100 times as much as the RC. Thus, the execution probability of the FC is over 100 times greater than that of the RC, i.e., $P_C$ (importance factor) is less than 0.01, or an estimated $P_{upp}$ is 0.01. Several observations can be drawn from this table:

• The fault density measured from test for the RC was only one-third that of the FC. This indicates that the test case distribution attempted to match the operational profile in terms of the FC and RC. By applying importance sampling to the operational profile to increase the number of RC test cases a greater number of failures would have been encountered during test and the removal of these faults would have reduced the number of failures in the operational phase. That is, stress testing of the RC such as redundancy management and exception handling code could achieve the highest benefits.

• If the acceleration factor is 100, the 42 RC failures incurred during the first year of operation would probably have been found in a few days of stress testing concentrated on the critical operations. To find the 32 faults responsible for the FC failures during the first operational year would probably have required almost a year of test since these functions were presumably accessed in operation as frequently as they were in test.

• The comparison of the FC and RC failures during the last four months (9 vs. 32) shows that the RC failures were dominant in the well tested software. The dependability estimated from the RC failures would thus be close to the actual dependability (based on all failures). Even for the first whole operational year, the dependability estimated from the 42 RC failures occurring during the year would be in the same order of that estimated from all the 75 failures of the year.

**Table 1. Characteristics of DSN Code**

| Code characteristic | FC | RC |
|---|---|---|
| Program size (KSLOC) | 185.1 | 144.3 |
| Number of faults found in test | 893 | 235 |
| Fault density measured from test | 4.82 | 1.63 |
| Failures during first year operation | 33 | 42 |
| Failures during last 4 months of year | 9 | 32 |

### 3.2 Importance Sampling from Operational Profile

As discussed in the previous subsection, the operational profile is partitioned into two sets: the *critical set* (critical and rarely executed operations) and the *regular set* which includes the other operations. Let $oc_1$, $oc_2$, ..., $oc_n$ be the disjoint operations in the critical set and $or_1$, $or_2$, ..., $or_m$ be the disjoint operations in the regular set. The occurrence

probabilities for these operations are denoted by $pc_1, pc_2, ..., pc_n, pr_1, pr_2, ..., pr_m$ and satisfy

$$\sum_{i=1}^{n} pc_i + \sum_{i=1}^{m} pr_i = P_C + P_R = 1 \quad (8)$$

where $P_C = \sum pc_i$ is the probability that the system is operating in the critical set (*importance factor*) and $P_R = \sum pr_i$ is the probability that the system is operating in the regular set. Normally,

$$P_C << P_R \quad (9)$$

Assume the software system has been tested extensively (but mostly by the regular set) before the start of the stress testing. As discussed later, random (or selective) testing and fault injection are feasible techniques to exercise operations in the critical set. For example, to test nuclear safety software, various challenges can be randomly generated by emulating emergency conditions. To test the fault detection and redundancy management software, various faults can be injected. In the operational profile, these operations typically have extremely low occurrence probabilities. Based on the failure biasing technique (which reduces variance on estimates) used in [10], we increase the occurrence probabilities for the operations in the critical set to $pc'_1, pc'_2, ..., pc'_n$ such that

$$pc'_i = \frac{pc_i}{P_C} \quad (10)$$

Then, we have

$$\sum_{i=1}^{n} pc'_i = \frac{1}{P_C} \sum_{i=1}^{n} pc_i = 1 \quad (11)$$

In a test set, a number of critical operations can be exercised. In general, for test set $O_S$, $k$ critical operations, $oc_{s1}, oc_{s2}, ..., oc_{sk}$, can be tested. Since these operations are disjoint, the probability that any of these operations occurs is the sum of individual probabilities for these operations. Thus, the likelihood ratio for $O_S$ is

$$\Lambda(O_S) = \frac{P(O_S)}{P'(O_S)} = \frac{pc_{s1} + pc_{s2} + ... + pc_{sk}}{pc'_{s1} + pc'_{s1} + ... + pc'_{sk}} = P_C \quad (12)$$

There are two basic types of stress testing: (1) *random or selective testing* and (2) *fault or error injection*. For the first type, testing is performed continuously by sampling from the critical set. For the second type, testing is done intermittently, normally injecting one fault each time. Estimation of failure rate is only meaningful for the first type of testing (methods to estimate failure rate are discussed in Section 3.4). Let $\lambda'_C$ denote the failure rate estimated from the first category of testing. By Equations (6) and (7), the failure rate for the critical set in the normal operation should be

$$\lambda_C = \Lambda(O_S)\lambda'_C = P_C\lambda'_C < P_{upp}\lambda'_C \quad (13)$$

Recall assumptions for this approach: failures of critical operations are the major threat to system dependability and critical operations are not frequently executed. If residual failures are not limited to critical set of operations (redundancy management, exception handling, etc.), failures of the regular set should also be counted. Since we have assumed hat the failure rate for the critical set is dominant in the late testing phase or early operational phase as supported by the data in Table 1, the failure rate estimated from the stress testing of the critical set would be close to (at least in the same order of) the total failure rate. However, to have a more conservative assessment, the following estimate can be used:

$$\lambda = \lambda_R + \lambda_C < 2\lambda_C \quad (14)$$

where $\lambda$ is the total failure rate and $\lambda_R$ is the failure rate for the regular set. Because of the dominance of $\lambda_C$, $\lambda$ is bounded by $2\lambda_C$. Combining the above two equations, an upper bound of $\lambda$ is

$$\lambda_{upp} = 2P_{upp}\lambda'_C \quad (15)$$

As shown in the second example discussed below, the new probabilities assigned to critical operations may not be proportional to the original probabilities as recommended in Equation (10) (a recommendation for reducing variance). The "inverted operational profile" in the example can be constructed in this way:

$$p'_i = \frac{p_{\max} - p_i}{\sum_i (p_{\max} - p_i)} \quad (16)$$

where $p_i$ is the original probability for operation $i$, $p_{max}$ is the maximum among $p_i$'s, and $p'_i$ is the new probability for operation $i$. In such a case, the likelihood ratio for a test set will not be the same as the importance factor. It has to be calculated based on Equation (12) for the test set.

For the second type of stress testing, fault/error injection, an important parameter to estimate is the fault recovery probability, $C$, which characterizes the fault tolerance ability of the system. A similar parameter for a safety system is the probability of successful response to an emergency demand, $P_S$ (the criticality of this parameter is discussed in Section 3.5). Methods to estimate $C$ or $P_S$ are discussed in Section 3.4.

*Examples*

Operational profile has been used in the stress testing of FASTAR, the AT&T's Fast Automated Restoration Platform, although the application did not relate it to dependability quantification: "The FASTAR operational profile showed that cable cuts happen very infrequently.

Restoration of cable cuts was certainly the most critical operation of the system. Since it was not feasible to execute the test program for a year to simulate the expected 10 cuts, we designed accelerated test runs. In the case of the Restoration Node Controller, load was accelerated by a factor of 24 during stability tests. This could execute a year's operational profile in the test environment within a matter of weeks." [35]

An interesting example is the use of the inverted operational profile that represents test cases from ultra-rare regions in fault injection case studies on software from NASA [45]. In case study 1, Yaw, serving as a small yawdamp controller for a Boeing 737 aircraft delivered to NASA for research purposes, was tested under fault injections. In case study 2, Autopilot, providing the autopilot controls for a Boeing 737 airplane, was tested under fault injections. For the original operational profile, the failure tolerance score is 0.43 for Yaw and 0.95 for Autopilot. For the inverted operational profile, the failure tolerance score is reduced to 0 for Yaw and 0.05 for Autopilot.

### 3.3 Stress Testing and Data Collection

In stress testing, it is intended to exercise rare conditions as much as possible and meanwhile, to collect necessary data from the testing for dependability analysis. Figure 2 shows a typical progression curve of failure types constructed based on the failure data from the final (Level 8) testing of Space Shuttle Avionics (SSA) software [18]. The scale changes midway along the horizontal axis to permit a better display of the events in the right tail. The stress testing addressed here is for the phase represented by this tail. The following paragraphs discuss several possible stress testing methods.

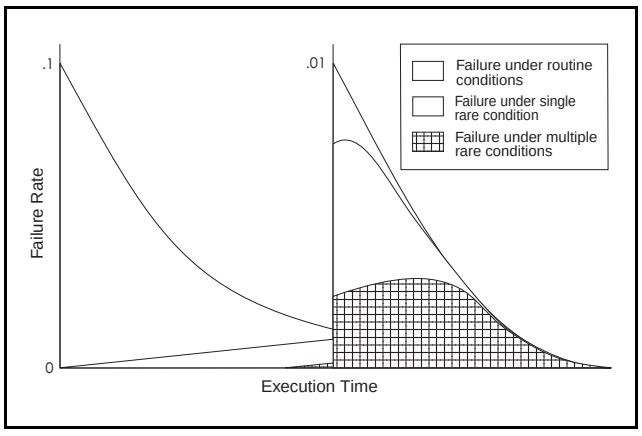


**Figure 2 Progression of Failure Types During Test**

*Random/Selective Sampling Testing*

This method is to test the target software by random or selective sampling from an input data set that is rich in opportunities for creating rare conditions. That is, the data set is chosen such that it is likely to produce operations in the critical set or rarely executed set of the operational profile. Random testing has long been proposed for use in the final testing stage of software to quantify its operational reliability [13]. This technique is economically feasible only where the correct results of a test can be easily identified.

The stability tests of air traffic control systems, the Advanced Automation System (AAS) [40] and the VSCS mentioned previously, are examples of random/selective sampling testing. A stability test is to exercise the target system with representative input data on a large configuration continuously for a number of hours (typically 72 to 120 hours). Although the duration of the test is not very long, due to the large configuration in which multiple copies of a software task are processing different input data, the accumulative execution time for the task can be as long as thousands of hours. In these tests, the execution of some critical operations (such as the switchover of air to ground communication channels discussed above) was accelerated, but by only a limited factor (10 for the switchover operation mentioned previously).

To quantify dependability using the importance sampling method, the following data need to be collected: (1) time spent on testing the critical set, (2) number of failures occurring in each component during the testing, (3) severity of each failure, and (4) if possible, time spent on the system recovery (channel switching, reconfiguration, or restart) from each failure. These data are necessary for estimating parameters such as failure rate, recovery rate, and coverage that are required by dependability models.

*Failure Mode Based Fault Injection*

Fault injection is an effective means to test software robustness, especially for fault tolerant software. Several techniques can be used to inject a fault or error to a software program such as: (1) modify the code segment of the program (fault injection) [21], (2) modify the data segment of the program (error injection) [21], and (3) robustness benchmark testing [37]. The failure mode based fault injection can be implemented with any of these techniques, while the error data injection to be discussed next is limited to the last two techniques.

In most cases, it is impossible to inject all possible faults because there are numerous types of faults and numerous ways a program can exercise faults. However, the effects (manifestations) of these faults are limited and can be characterized by a number of failure modes [7, 37]. Some typical failure modes are listed below:

• Stop — also referred to as "crash" or "task termination", the result of an error condition that occurs when software faults cause the task to cease processing.

• Hang — the result of an error condition that occurs

when software faults cause the task to cease useful processing, yet continue regular processing and memory demands.

• Delayed output — the result of an error condition that occurs when software faults cause required outputs of the task to be delivered outside of acceptable time ranges.

• Invalid output — the result of an error condition that occurs when software faults cause the task to deliver erroneous outputs.

The failure mode based fault injection is to generate a failure mode by injecting faults. This approach is suitable for fault tolerant software and has been used to test the AAS air traffic control system [12]. The approach is as follows: For each software module, all possible failure modes are first identified. Then for each identified failure mode, a number of typical faults (e.g., 5) are designed and implemented in the module to induce the failure mode. After the activation of an injected fault, the software module is monitored and the following data are collected for parameter estimation: (1) result of the failure recovery (success or non-success) which is useful for estimating the "coverage" parameter used in dependability models discussed later, and (2) time spent on the failure recovery which is useful for estimating failure recovery rate.

*Error Data Injection*

Recall the VSCS failure case discussed in Section 3.1. This event was the result of three rare conditions: a memory chip fault, hardware memory error detection function disabled, and software defects in data consistency checking. A lesson learned from this event is that data errors can be caused by hardware faults and software should prevent catastrophic failures from these errors. The Ariane disaster was also triggered by faulty data and then the lack of protection code led to the failure, except that the faulty data were generated by another faulty software module. A recent study classified the causes for software failures as three categories: erroneous input data, faulty code, or a combination of the two [44]. When the faulty code is rarely executed, the third category generates typical rare conditions as shown in the VSCS and Ariane failure events. Error injection into data is to target this issue.

One way to inject erroneous data is to change system internal state. It has been shown that many hardware faults, such as CPU, memory and network faults, can be emulated by altering memory elements [24]. Thus, alternation of the data segment of the program in memory is a representative approach to injecting software data errors caused by hardware faults. Data items to which errors will be injected can be randomly selected or determined in advance. In the latter case, if critical data items have been identified, values out of expected ranges can be assigned to these data items as error injection.

Another way to inject erroneous data is to corrupt input data. This approach has been applied to a nuclear safety software system by corrupting simulated sensor signals. “In this case study, safety monitoring code was found to be faulty and not functioning properly. Fault injection easily forced the software to ‘think’ that a hazardous event was occurring when it was not, and vice versa. Had the safety monitoring code been working properly, fault injection would not have been able to make this happen.” [44]

The *robustness benchmark testing* introduced in [37] can be viewed as a particular kind of error data injection. A robustness benchmark is a program designed to measure how an operating system or other commonly used software reacts to erroneous inputs. In [37], several primitive benchmarks were developed on the UNIX system. Each primitive benchmark targets a system functionality, such as file management and memory access, by exercising system calls with erroneous input data (e.g., open a file with a too long name) or with destructive manners (open too many files). A robust system should be able to tolerate these erroneous inputs or usages by reporting errors and avoiding system crashes. Unfortunately, none of the tested commercial systems (DEC, Sun, etc.) showed a high degree of fault tolerance.

Since error data injection can induce the same failure modes discussed above [37], similar to the failure mode based fault injection approach, it can furnish the estimation of the coverage and failure recovery time parameters.

### 3.4 Parameter Estimation

Based on the data collected from stress testing, various parameters can be estimated. These parameters will then be used in the dependability modeling and evaluation discussed in the next subsection. The most frequently used parameters can be estimated using the following methods.

Assume $n$ failures ($n$ could be 0) have been observed in performing test on the critical set $O_S$ and the execution time for $O_S$ is $T'$. A failure rate upper bound can then be estimated by [25]

$$\lambda'_{upp} = \frac{\chi^2_{1-\alpha;2n+2}}{2T} \quad \textbf{(17)}$$

where $\alpha$ is the significance level ($1-\alpha$ is the confidence level) and $\chi^2_{1-\alpha;2n+2}$ determines a Chi-square probability such that $P(X_{2n+2} < \chi^2_{1-\alpha;2n+2}) = 1-\alpha$. When $n$ is zero, i.e., no failure has been observed, the equation also applies and in this case, it is equivalent to the following estimator given in [40]:

$$\lambda'_{upp} = \frac{-\ln(\alpha)}{T} \quad \textbf{(18)}$$

Based on methods discussed in Section 3.2, $\lambda'_{upp}$ can be converted to $\lambda_{upp}$.

If the parameter of interest is the probability of successful response to a demand of emergency handling or channel switching due to a failure, i.e., the parameter $C$ or $P_S$ discussed in Section 3.2, the following equation can be used to estimate a lower bound [25]:

$$C_{low} = \frac{1}{1 + \frac{n-s+1}{s} F_{1-\alpha;2(n-s)+2;2s}} \quad \textbf{(19)}$$

where $n$ is the number of trials, $s$ is the number of successful trials, $\alpha$ is the significance level, and $F$ represents the F statistical distribution.

If $s = n$, the estimated probability would be 1 (a 100% coverage), which is usually not representative of the likely true state because of imperfect fault detection mechanisms, defects in redundancy management or safety management software, and common mode failures. Thus, a value less than 1 is preferred in conservative dependability modeling. In such a case, a lower bound of the probability can be estimated by [40]

$$C_{low} = \alpha^{\frac{1}{n}} \quad \textbf{(20)}$$

### 3.5 Dependability Modeling and Evaluation

A system can be modeled hierarchically: system level, subsystem level, and lower levels of modeling as required. This hierarchical modeling approach is recommended because (1) it reduces model complexity and (2) it facilitates identifying problem areas (by comparing results from submodels). Different levels of modeling may use different model structures, depending on the architecture and behavior of the modeled system. If failures of components are relatively independent in the modeled system, the simpler structures of reliability block diagrams and *k*-out-of-*n* models suffice. This is usually the case for the high level modeling in which subsystem failures can be considered independent. If the failure of a component may affect other components, or a reconfiguration is involved upon a component failure, Markov chains and other models (e.g., Petri net) that can account for interactions between modeled components are preferred. This is usually the case for the low level modeling or any modeling where failure correlations among components are strong.

This modeling approach has been applied to both continuously operating real-time systems and on-line protection systems. In [40], an availability model was developed from the software task level for an air traffic control system and stability test data were used to estimate model parameters. In [41], a nuclear power plant safety model was developed based on a real protection system and its early field failure data for sensitivity analysis. A user-friendly software tool called MEADEP [41, 42] has also been developed for data processing, parameter estimation, graphical model construction, and dependability evaluation, based on structured failure reports generated for systems in operation.

Figure 3 is a layout of MEADEP. The *Data PreProcessor* (DPP) module, interacts with the user to convert source data to MEADEP internal data. The source data can be manually generated structured trouble reports or computer generated event logs. The *Data Editor and Analyzer* (DEA) module is used to edit internal data and to perform statistical analysis on the data. Parameter values estimated from the data by this module can be inserted into the text modeling file generated by another module, the *Model Generator* (MG). The MG module provides a graphical user interface for the user to draw model diagrams and then to generate, from the diagrams, a text modeling file that contains model specifications suitable for solution. The *Model Evaluator* (ME) module produces results based on the model specifications and parameters in the text modeling file. All modules are integrated with the *Graphical User Interface* (GUI) featuring menus, dialogs, graphical input and output, and extensive on-line help information. Further details on MEADEP are discussed in [41, 42].

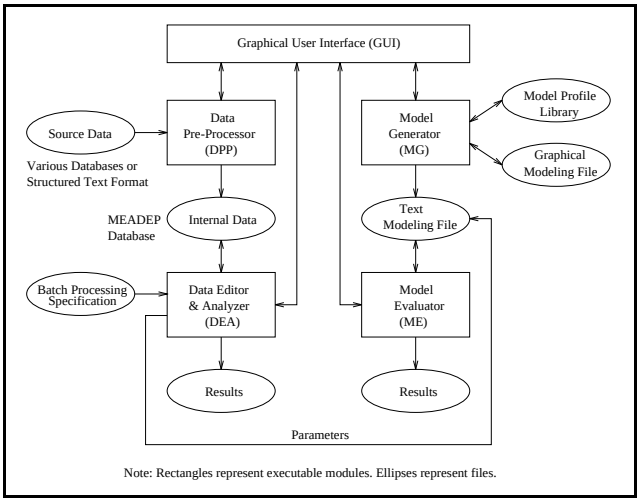


**Figure 3 Layout of MEADEP**

## 4. Conclusion

In this paper, we described an approach that combines the operational profile, rare conditions, importance sampling, stress testing, and measurement-based dependability

evaluation in the late testing phase or early operational phase, to quantify dependability for a category of critical software constrained by the following conditions: (1) the operational profile for the software is identifiable; (2) critical operations constitute only a small part of the operational profile; and (3) failures of critical operations are the major threat to system dependability. Feasible methods to implement the approach have been identified and discussed based on real data, including guidelines in constructing operational profiles, determination of the likelihood ratio, stress testing methods, parameter estimation methods, and dependability modeling methods. Not only can the approach be used to quantify dependability for critical software systems in which rare conditions are the major threat to the system dependability, but it can also contribute significantly to enhancing the quality of the software when deficiencies found in stress testing are resolved. Further efforts need to be made in developing tools to assist the implementation of the approach and in applying the approach to real projects.

## References


[1] A. A. Abdel-Ghaly, P. Y. Chan, and B. Littlewood, "Evaluation of Competing Software Reliability Predictions," *IEEE Transactions on Software Engineering*, Vol 12, No. 9, Sept. 1986, pp. 950-967.

[2] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. C. Fabre, J. C. Laprie, E. Martins, and D. Powell, "Fault Injection for Dependability Validation: A Methodology and Some Applications," *IEEE Transactions Software Engineering*, Vol. 16, No. 2, Feb. 1990, pp. 166-182.

[3] American National Standards Institute, *Recommended Practice for Software Reliability*, ANSI/AIAA R-03-1992, 1993.

[4] A. Avizienis, "The N-Version Approach to Fault-Tolerant Software," *IEEE Transactions on Software Engineering*, Vol. 11, No. 12, Dec. 1985, pp. 1491-1501.

[5] D. Avresky, J. Arlat, J. C. Laprie, and Y. Crouzet, "Fault Injection for Formal Testing of Fault Tolerance," *IEEE Transactions on Reliability*, Vol. 45, No. 3, Sept. 1996, pp. 443-455.

[6] R. W. Butler and G. B. Finelli, "The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software," *IEEE Transactions on Software Engineering*, Vol. 19, No. 1, Jan. 1993, pp. 3-12.

[7] J. H. Barton, E. W. Czeck, Z. Z. Segall and D. P. Siewiorek, "Fault Injection Experiments Using FIAT," *IEEE Transactions on Computers*, Vol. 39, No. 4, April 1990, pp. 575-582.

[8] R. Chillarege and N. S. Bowen, "Understanding Large System Failures — A Fault Injection Experiment," *Proceedings of the 19th International Symposium on Fault-Tolerant Computing*, June 1989, pp. 356-363.

[9] G. Choi and R. K. Iyer, "Wear-Out Simulation Environment for VLSI Designs," *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*, June 1993, pp. 320-329.

[10] A. E. Conway and A. Goyal, "Monte Carlo Simulation of Computer System Availability/Reliability Models," *Proceedings of the 17th International Symposium on Fault-Tolerant Computing*, June 1987, pp. 230-235.

[11] A. N. Crespo, P. Matrella and A. Pasquini, "Sensitivity of Reliability Growth Models to Operational Profile Errors," *Proceedings of the Seventh International Symposium on Software Reliability Engineering*, Nov. 1996, pp. 35-44.

[12] T. R. Dilenno, D. A. Yaskin and J. H. Barton, "Fault Tolerance Testing in the Advanced Automation System," *Proceedings of the 21st International Symposium on Fault-Tolerant Computing*, June 1991, pp. 18-25.

[13] J. W. Duran and S. C. Ntafos, "An Evaluation of Random Testing," *IEEE Transactions on Software Engineering*, Vol. 10, No. 4, July 1984, pp. 438-444.

[14] D. E. Eckhardt, A. K. Caglayan, et al., "An Experimental Evaluation of Software Redundancy as a Strategy for Improving Reliability," *IEEE Transactions on Software Engineering*, Vol 17, No. 7, July 1991, pp. 692-702.

[15] W. Farr, "Software Reliability Modeling Survey," *Handbook of Software Reliability Engineering*, M. R. Lyu, Editor, McGraw-Hill, New York, 1996, pp. 71-117.

[16] A. Goyal, P. Shahabuddin, P. Heidelberger, V. F. Nicola, and P. W. Glynn, "A Unified Framework for Simulating Markovian Models of Highly Dependable Systems," *IEEE Transactions on Computers*, Vol. 41, No. 1, Jan. 1992, pp. 36-51.

[17] J. Gray, "A Census of Tandem System Availability Between 1985 and 1990," *IEEE Transactions on Reliability*, Vol. 39, No. 4, Oct. 1990, pp. 409-418.

[18] H. Hecht, "Rare Conditions — An Important Cause of Failures," *Proceedings of the Eighth Annual Conference on Computer Assurance*, June 1993, pp. 81-85.

[19] H. Hecht and P. Crane, "Rare Conditions and Their Effect on Software Failures," *Proceedings of the Annual Reliability and Maintainability Symposium*, Jan. 1994, pp. 334-337.

[20] M. C. Hsueh and R. K. Iyer, "A Measurement-Based Model of Software Reliability in a Production Environment," *Proceedings of the 11th Annual International Computer Software & Applications Conference*, Oct. 1987, pp. 354-360.

[21] R. K. Iyer and D. Tang, "Experimental Analysis of Computer System Dependability," *Fault-Tolerant Computer System Design*, D. K. Pradhan (ed.), Prentice Hall PTR, NJ, Feb. 1996, pp. 282-393.

[22] H. Kahn and A. W. Warshall, "Methods of Reducing Sample in Monte Carlo Computations," *Journal of the Operations Research Society of America*, Vol. 1, No. 5, 1953, pp. 263-278.

[23] K. Kanoun, M. Kaaniche and J. C. Laprie, "Qualitative and Quantitative Reliability Assessment," *IEEE Software*, March/April 1997, pp. 77-87.

[24] W-L. Kao, R. K. Iyer, and D. Tang, "FINE: A Fault Injection and Monitor Environment for Tracing the UNIX System Behavior under Faults," *IEEE Transactions on Software Engineering*, Vol. 19, No. 11, Nov. 1993, pp. 1105-1118.

[25] D. Kececioglu, *Reliability and Life Testing Handbook*, Vol. 1 & 2, PTR Prentice Hall, Englewood Cliffs, NJ, 1993.

[26] J. C. Knight and N. G. Leveson, "An Experimental Evaluation of the Assumptions of Independence in Multiversion Programming," *IEEE Transactions on Software Engineering*, Vol. 12, Jan. 1986, pp. 96-109.

[27] D. R. Kubn, "Sources of Failures in the Public Switched Telephone Network," IEEE Computer, April 1997, pp. 31-36.

[28] B. Littlewood, "Predicting Software Reliability," *Phil. Trans. Roy. Soc. London*, 1989, pp. 95-117.

[29] I. Lee and R. K. Iyer, "Analysis of Software Halts in Tandem System," *Proceedings of the Third International Symposium on Software Reliability Engineering*, Oct. 1992, pp. 227-236.

[30] I. Lee, D. Tang, R. K. Iyer and M. C. Hsueh, "Measurement-Based Evaluation of Operating System Fault Tolerance," *IEEE Transactions on Reliability*, Vol. 42, No. 2, June 1993, pp. 238-249.

[31] I. Lee and R. K. Iyer, "Software Dependability in the Tandem GUARDIAN System," *IEEE Transactions on Software Engineering*, Vol. 21, No. 5, May 1995, pp. 455-467.

[32] J. D. Musa, A. Iannino and K. Okumoto, *Software Reliability: Measurement, Prediction, Application*, McGraw-Hill Book Company, 1987.

[33] J. D. Musa, "The Operational Profile in Software Reliability Engineering: An Overview," *Proceedings of the Third International Symposium on Software Reliability Engineering*, Oct. 1992, pp. 140-154.

[34] J. D. Musa, "Sensitivity of Field Failure Intensity to Operational Profile Errors," *Proceedings of the Fifth International Symposium on Software Reliability Engineering*, Nov. 1994, pp. 334-337.

[35] J. Musa, G. Fuoco, N. Irving, D. Kropfl, and B. Juhlin, "The Operational Profile," *Handbook of Software Reliability Engineering*, M. R. Lyu, Editor, McGraw-Hill, New York, 1996 pp. 167-216.

[36] Z. Segall, et al., "FIAT — Fault Injection Based Automated Testing Environment," *Proceedings of the 18th International Symposium on Fault-Tolerant Computing*, June 1988, pp. 102-107.

[37] D. P. Siewiorek, J. J. Hudak, B. H. Suh, Z. Segall, "Development of a Benchmark to Measure System Robustness," *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*, June 1993, pp. 88-97.

[38] M. Sullivant and R. Chillarege, "Software Defects and Their Impact on System Availability — A Study of Field Failures in Operating Systems," *Proceedings of the 21st International Symposium on Fault-Tolerant Computing*, June 1991, pp. 2-9.

[39] D. Tang and R. K. Iyer, "Analysis of the VAX/VMS Error Logs in Multicomputer Environments — A Case Study of Software Dependability," *Proceedings of the Third International Symposium on Software Reliability Engineering*, Oct. 1992, pp. 216-226.

[40] D. Tang and M. Hecht, "Evaluation of Software Dependability Based on Stability Test Data," *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, June 1995, pp. 434-443.

[41] D. Tang, M. Hecht, J. Agron, J. Miller and H. Hecht, "Engineering Oriented Dependability Evaluation: MEADEP and Its Applications," *Proceedings of the Pacific Rim International Symposium on Fault-Tolerant Systems*, Taipei, Taiwan, Dec. 1997.

[42] D. Tang, M. Hecht, J. Handal and L. Czekalski, "MEADEP and Its Applications in Evaluating Dependability for Air Traffic Control Systems," *Proceedings of the 1998 Annual Reliability and Maintainability Symposium*, Anaheim, California, Jan. 1998.

[43] J. Voas, "Software Fault Injection: Growing 'Safer' Systems," *Proceedings of the IEEE Aerospace Conference*, Snowmass, Co., Feb. 1997.

[44] J. Voas, G. McGraw, L. Kassab and L. Voas, "A 'Crystal Ball' for Software Liability," *IEEE Computer*, June 1997, pp. 29-36.

[45] J. Voas, F. H. Charron, G. McGraw, K. Miller and M. E. Friedman, "Predicting How Badly 'Good' Software Can Behave," *IEEE Software*, Vol. 14, No. 4, July/Aug. 1997, pp. 73-83.