Reliability Measurement, Analysis, and Improvement for Large Software Systems*

Jeff Tian
Dept. of Computer Science and Engineering
Southern Methodist University
Dallas, Texas 75275
E-mail: tian@seas.smu.edu
Home page: http://www.seas.smu.edu/~tian

Abstract

This article surveys recent developments in software reliability engineering, and presents results in measuring test activities, analyzing product reliability, and identifying high risk areas for focused reliability improvement for large software systems. Environmental constraints and existing analysis techniques are carefully examined to select appropriate analysis techniques, models, implementation strategies, and support tools suitable for large software systems. The analysis and modeling activities include measure definition, data gathering, graphical test activity and defect tracking, overall quality assessment using software reliability growth models, and identification of problematic areas and risk management using tree-based reliability models. Various existing software tools have been adapted and integrated to support these analysis and modeling activities. This approach has been used in the testing phase of several large software products developed in the IBM Software Solutions Toronto Laboratory and was demonstrated to be effective and efficient. Various practical problems and solutions in implementing this strategy are also discussed.

Keywords: large software systems, reliability, testing, risk identification and management, tools and environments.

Contents

*The final version of this paper was published in *Advances in Computers, Vol.46: The Engineering of Large Systems.* M. V. Zelkowitz, editor, pp.159-235, Academic Press, 1998.

1 Overview and Organization

Reliability captures the likelihood of software failures for a given time period or a given set of input values under a specific environment, and can be used as the primary quality measure during software development, particularly in the testing phase. One fundamental assumption in reliability analysis and modeling is the in-feasibility of cost-effective prevention, detection and elimination of *all* defects in software products [8, 35, 34, 40]. There is also a general trend of diminishing returns in quality improvement for resource spent on testing as more and more defects are removed from a software system [10, 40]. The need for the optimal combination of high quality and low cost requires us to collect appropriate measurement data, analyze them using various mathematical and statistical models to assess product quality, and make optimal resource allocation and product release decisions. These measurement and analysis activities can be integrated into the overall software process measurement and improvement framework [26] to help both developers and testers to detect and remove potential defects, to improve the development process to prevent injection of similar defects, and to manage risk better by planning early for product support and service.

This article examines the need for reliability analysis and improvement for large software systems in the component and system testing phases of the software development life-cycle. Various models are used to track reliability growth and offer an overall assessment of product quality [60]. Other analyses include identification of specific problematic (or high risk) areas and deriving remedial actions to solve or alleviate the problems identified [55]. To facilitate easy understanding and interpretation of data and results, visualization techniques and graphical representations are used to examine measurement data, track progress, and present modeling results. Software tools are used and integrated to support data collection, reliability analysis and

SRE:	software reliability engineering
SRM:	software reliability model(s)
SRGM:	software reliability growth model(s)
SRGM-DC:	software reliability growth model(s) based on data clustering (partitioning)
IDRM:	input domain reliability model(s)
TBM:	tree-based modeling, or tree-based model(s)
TBRM:	tree-based reliability model(s) that integrate SRGMs and IDRMs using TBM

Table 1: Acronyms

modeling, and result presentation [64]. This approach has been demonstrated to be effective, efficient, and easily accepted by the development community [56]. So far, it has been used in several large commercial software products developed in the IBM Software Solutions Toronto Laboratory. Various practical problems and solutions in implementing and deploying this approach are also discussed.

Acronyms used throughout this article are listed in Table 1. Section 2 provides a survey of existing reliability analysis techniques and several commonly used software reliability models (SRMs), including both the time domain software reliability growth models (SRGMs) and the input domain reliability models (IDRMs), and discusses their common assumptions and applicability in general terms. Section 3 describes testing environment for large software systems and the specific needs for quality assessment and improvement under such an environment. Section 4 examines test activities and workload measurements and presents some recent results applying various SRGMs in assessing and predicting reliability for large software systems. Section 5 outlines a tree-based reliability modeling (TBRM) technique and its usage in identifying high risk (low reliability) areas for focused reliability improvement. Section 6 presents SRGM-DC, a new type of SRGMs based on data clustering analysis and its applications. Section 7 describes implementation and tool support for various reliability analyses described in this article. Section 8 summarizes the article and offers some future perspectives.

2 Techniques and Models for Analyzing Software Reliability

Throughout the testing process, various measurement data need to be gathered and analyzed to track progress, assess product quality and reliability, manage resource allocation and product release, and optimize testing and development processes [26]. This section surveys existing reliability measurements and analysis techniques that are applicable to the testing process.

2.1 Software Quality, Reliability, and Analyses

Software quality can be defined and measured from various different views to reflect different concerns and aspects of "quality" [31]. In practice, it is generally defined either from a customer view based on external behavior of the software product or from a development view based on internal product attributes, and measured accordingly. According to Kan et al., "Quality is conformance to customer's expectations and requirements" [29]. This definition emphasizes the central role of customers, who are the final arbiter of how "good" a software product is. These customer expectations and requirements are represented by the product specifications during product development. These specifications can be formally verified during specification and implementation processes using various formal verification techniques [75] or dynamically verified through different types of testing [8]. Various quality measures from the developer's perspective can be defined based on these specifications.

Key to the conformance of customer's expectations and product specifications is the concept of failure, fault and error: A *failure* is a behavioral deviation from the user requirement or specification, a *fault* is the underlying cause within a software system that causes certain failure(s), while an *error* refers to human mistakes or misconceptions that result in faults being injected into software systems [1, 2]. Failure is a

dynamic, behavioral definition, which is closer to the customer's view, while fault is a static, code-based definition, closer to the developer's view. Both failures and faults are often referred to as *defects*, when the distinction between the cause (faults) and effect (failures) is not critical [34].

Various defect measures, such as defect count and defect density, are commonly used as direct quality measures in many software development organizations. However, such measures are defined from a developer's view, and it is hard to predict post-release defects based on observations of pre-release defects because the differences in environments. Recently, the trend has been to move toward quality measures that are more meaningful to the customer. Software reliability and related measures, such as failure intensity or MTBF (mean-time-between-failure), have gained wide acceptance [34, 40].

Generally speaking, a software system is said to be "reliable" if it performs its specified functions correctly over a long period of time or under a wide variety of usage environments and operations [24, 40, 54]. Reliability is a failure-centered quality measure that views the software system as a whole from a customer perspective. *Software reliability engineering* (SRE) is concerned with the assessment, prediction, and management of the reliability of software systems. SRE activities are generally concentrated on the system testing stage, because only by then are all the functions available to the customer available for testing.

One fundamental assumption in SRE is the existence of faults and in-feasibility of exhaustive testing that can be used to expose and remove all the faults in software systems. As a result, failure-free operations cannot be guaranteed, but only assured statistically based on past observations. A *software reliability model* (SRM) provides a systematic way of assessing and predicting software reliability. Based on different assumptions regarding the characteristics of the software systems and fault exposure, various reliability models have been proposed and used to analyze software reliability. There are two types of software reliability models:

- Time domain *software reliability growth models* (SRGMs): These models analyze time-indexed failure data to assess the software's ability to perform correctly over a specific time period under a given operational environment.
- *Input domain reliability models* (IDRMs): These models analyze input states and failure data to assess the software's ability to perform correctly for different input values.

These two approaches provide two different perspectives of reliability. The time domain approach stresses the assessment and prediction of the overall reliability. The input domain approach provides valuable information that can be used to thoroughly test software products due to the use of well-defined input states. However, they are generally used disjointly in practice. Recently, an integrated approach that combines some strengths of these two approaches was developed in [55, 56] to provide analysis results that can be used to assess the reliability of software products as well as to guide reliability improvement.

2.2 Testing Techniques and Operational Profiles

Testing of small programs or small units of large programs (unit testing) is usually performed by the program developer with complete knowledge of the implementation details of the small programs or program units. Such testing is generally referred to as white-box (or structural) testing, which usually strives for complete coverage of all possible states, execution paths etc. [8]. There is no separation of duties between failure detection and quality assurance on the one side and problem tracking and fault removal on the other. However, for overall testing (or system testing) of large software systems, this approach becomes impractical because of the combinatorial explosion of possible states and execution paths. Professional testers are needed, black-box (or functional) testing techniques are often used, and automated testing tools become more of a necessity than luxury [9]. To achieve effective failure detection and to provide statistically verifiable means of demonstrating the product's reliability, one must thoroughly test the software with a variety of input to cover important functions and relations. The concept of *operational profiles*, collections of functions and their usage frequencies by some target user group [38, 39], has gained wide usage in system testing recently.

Operational profiles can be used to help testing teams to cover a maximal number of situations in which the program will be used by customers, within practical limits due to budget and/or schedule constraints.

Most of the SRE activities also assume the availability and usage of operational profiles in testing, so that modeling results from this in-process reliability analysis can be extrapolated to predict in-field software reliability. Operational profiles define the usage environment, under which software reliability is defined and analyzed. The accuracy of reliability analysis result is affected by the accuracy of operational profiles [17]. However, a recent study demonstrated that the predictive accuracy of reliability models is not heavily affected by errors in the estimate of the operational profiles [43], thus even rough operational profiles can provide valuable information for software testing and reliability analyses. Recently, new concepts in test selection and design for testability have also been developed by various researchers for use in conjunction with operational profiles for effective testing [39, 70].

There are several way to capture operational profiles, including: 1) actual measurement in customer settings, 2) user or customer surveys, and 3) usage information gathering in conjunction with software requirement information gathering [38]. The first method is the most accurate, but also the most costly, and may also be problematic because it has to deal with legal issues and problems concerning intellectual properties. The latter two are less costly, but the information captured may not be accurate or detailed enough for us to develop operational profiles directly. However, such surveys and information gatherings can be used as a general guide to the overall testing process to ensure that at least the high level functions that are important to customers or frequently used by customers can be covered by testing [60]. A recent study also pointed out that a controlled laboratory environment, where customers are invited to product pre-release validations, may yield valuable data for the construction of operational profiles [33]

The results from measurements or surveys of customer usage can be organized into structured forms and used to derive test cases and generate test workloads. Commonly used models for organizing such information include 1) flat profiles which lists end-to-end user operations and associated probabilities such as in [38]; and 2) individual units of operations and associated transition probabilities among the units that form a Markov chain [3, 73]. Test cases and test workload can be generated from these structured operational profile information according to various rules and criteria to cover important functions frequently used by target customers.

2.3 Defining and Measuring Reliability in the Time Domain

In the time domain approach, the reliability of a software system is defined to be the probability of its failure-free operations for a specific duration under a specific environment [1, 22, 40]. Reliability is usually characterized by hazard and reliability functions. The hazard function (or hazard rate) $z(t)$ is defined as:

$$z(t)\Delta t \;=\; P\{t < T < t + \Delta t \mid T > t\}$$

where T marks the failure time, P is the probability function, and $z(t)\Delta t$ gives the probability of failure in time interval $(t, t+\Delta t)$, given that the system has not failed before t. The reliability function $R(t)$ is defined as:

$$R(t) \;=\; e^{-\int_0^t z(x)dx}$$

which gives the probability of failure free operations in the time interval (0, t). MTBF (mean time between failure) is commonly used as a measure of reliability for its intuitiveness. MTBF can be calculated as:

$$\text{MTBF} \;=\; \int_0^\infty R(x)dx$$

In practical applications, comparing to other reliability measures, the measure MTBF is easy to interpret and directly meaningful to customers as well as software managers, developers, and testers.

Various measurement data are necessary for model fitting and usage. There are three key elements to time domain reliability measurement: failure, time, and usage environment. The key to failure measurement is consistency in failure definition and data interpretation. The environment is generally assumed to be similar to the actual customer usage environment, so that the analysis results can be directly extrapolated to the likelihood of in-field product failures [39]. For time measurement, there are various alternatives, falling into two categories:

Label	Reference	Model
JM	[28]	Jelinski-Moranda de-eutrophication model
Sho	[51]	Shooman model
Geo	[36]	Moranda's geometric de-eutrophication model
GO	[25]	Goel-Okumoto NHPP model
S	[74]	Yamada-Ohba-Osaki S-shaped NHPP model
GP	[46]	Schafer *et al.* generalized Poisson model
SW	[47]	Schick-Wolverton model
BMB	[13]	Brooks-Motley model, binomial variation
BMP	[13]	Brooks-Motley model, Poisson variation
Musa	[37]	Musa's basic execution time model
MO	[41]	Musa-Okumoto logarithmic Poisson execution time model
Sch	[48]	Schneidewind model
LV	[32]	Littlewood-Varrel Bayesian model

Table 2: Labels and references for some commonly used SRGMs

- *Usage independent time measurement*: Only the failures are marked, either by the actual failure time or by failures loosely associated with a time period. Information about software usage is ignored. The commonly used calendar time and wall-clock time measurements fall into this category.
- *Usage dependent time measurement*: Only the time when software is used is counted, and the usage during testing is usually used to normalize the time measurement. For example, execution time measurement counts only the time when the software system is actually used [40]. Time can also be measured as some test activity count in a logical sequence, such as time-ordered test runs in [60] or transactions in [62].

If software usage is somewhat constant, usage independent time measurements are usable for reliability modeling. Otherwise, various test activity or workload measurements need to be taken for reliability modeling. Based on different time measurements and different assumptions about the underlying faults and the fault-failure linkages, different reliability growth models have been proposed and used [1, 22, 24, 40]. One underlying assumption common to all models is that the reliability grows as failure-causing faults are removed from the software, hence the name reliability *growth* models. Several SRGMs are listed in Table 2 and briefly described below:

De-eutrophication Models

De-eutrophication models link failure probability to the number of defects remaining in the current system in a functional form, thus capture reliability growth (or de-eutrophication) in testing as a result of defect observations and removals. In the Jelinski-Moranda model [28], one of the earliest and most widely used models, chance of failure for unit time is *proportional* to the number of defects remaining in the current system. That is, the hazard rate z_i for the i-th failure is:

$$z_i = \phi(N - (i-1))$$

where N is the total number of defects at the beginning of testing (i.e., before discovering the first failure), and ϕ a proportionality constant for the model. The hazard rate between successive failure observations remains constant, and the discovery and removal of each defect contribute the same to the hazard rate reduction. Therefore, the failure rates over successive failures form a step function of time, with uniform downward steps at corresponding failure observations.

The model proposed by Shooman [51] is essentially the same as the Jelinski-Moranda model above, but with different parameters and uses period failure count data instead of time-between-failure data. Another

similar de-eutrophication model is the geometric model by Moranda [36], with

$$z_i = z_0 \phi^{(i-1)}$$

where $\phi < 1$. The hazard function over the failure intervals forms a downward geometric sequence. The geometric model captures the slow-down effect in hazard rate reduction as more and more defects are discovered and fixed.

Generic NHPP Model and Variations

The failure arrivals can be viewed as a stochastic process and analyzed accordingly [30]. The most commonly used such process is the non-homogeneous Poisson process (NHPP), with the number of failures $X(t)$ for a given time interval $(0,\ t)$ prescribed by the probability $P[X(t) = n]$ as:

$$P[X(t) = n] = \frac{[m(t)]^n e^{-m(t)}}{n!}$$

where $m(t)$ is the *mean function*, and the failure rate $\lambda(t)$ (used in place of $z(t)$ in such situations) is the derivative of $m(t)$, i.e., $\lambda(t) = m'(t)$. Different choices of the mean function $m(t)$ can be used to model different failure arrival patterns. Two specific variations of the NHPP models used in this article are:

1. *Goel-Okumoto model* (also known as the exponential model) [25] is an NHPP model with

   $$m(t) = N(1 - e^{-bt})$$

   where N (estimated total defects) and b are constant. This model is the continuous equivalent to the discrete Jelinski-Moranda model.

2. *S-shaped reliability growth model* [74] is another NHPP model with

   $$m(t) = N(1 - (1 + bt)e^{-bt})$$

   where N (estimated total defects) and b are constant. This model better describes the commonly observed patterns of failure arrivals consisting of three phases: a initial slow start, a middle section characterized by numerous failure observations and fault removals, and followed by a slow down (saturation) of failure arrivals.

Generalized Poisson Model and Variations.

The generalized Poisson model [46] is a generalization of Jelinski-Moranda model and several related models. In this model, the distribution of the number of failures f_i for the i-th interval with test length t_i is a Poisson process, with the mean function $m_i(t_i) = E(f_i)$ specified as:

$$m_i(t_i) \ = \ \phi \ [N - M_{i-1}] \ g_i(t_1, t_2, \ldots, t_i)$$

where N is the estimated total defects, M_i the defects removed after i-th period, and ϕ the Poisson constant. Three notable variations to this model are:

1. *Jelinski-Moranda model* [28], with $f_i = 1$ and $g_i = t_i$.

2. *Schick-Wolverton model* [47], with $f_i = 1$ and $g_i = \frac{t_i^2}{2}$. The hazard rate increases with time in this model, i.e., $\lambda_i = \phi \ [N - M_{i-1}] \ t_i$. Therefore, chance for failure observations increases more than linearly with time.

3. Generalized Poisson model with estimated power term α, i.e, $g_i = (t_i)^\alpha$ [46]. This generalized model is data driven, and can fit to various different failure arrival patterns, from sub-linear ($\alpha < 1$), linear ($\alpha = 1$, i.e., Jelinski-Moranda model), to super-linear ($\alpha > 1$, including Schick-Wolverton model, with $\alpha = 2$, and others).

Brooks-Motley Model

The Brooks-Motley model [13] has two variations according to whether a Poisson or a binomial distribution of failure observations n_i over all possible $\{X\}$ for i-th period of length t_i is assumed:

$$P[X = n_i] = \begin{cases} \frac{(N_i \phi_i)^{n_i} e^{-N_i \phi_i}}{n_i!}, & \phi_i = 1 - (1 - \phi)^{t_i} \quad (Poisson) \\ \binom{N_i}{n_i} q_i^{n_i} (1 - q_i)^{N_i - n_i}, & q_i = 1 - (1 - q)^{t_i} \quad (binomial) \end{cases}$$

where N_i is the estimated number of defects at the beginning of i-th period; q and ϕ are the binomial and Poisson constants respectively. Unlike other models listed here that all assume that the full product is tested, the Brooks-Motley model allows for only a part of the product to be tested, making it the only choice under specific situations.

Execution Time SRGMs

Basic execution time model by Musa [37] is essentially the same as Jelinski-Moranda model, but with the emphasis of using CPU-execution time as the time measurement. This model also include an predictive element, enabling the user of this model to estimate model parameters from product and process characteristics even before actual failures are observed. Logarithmic execution time model by Musa and Okumoto [41] is a variation of NHPP model with

$$m(\tau) = \frac{1}{\theta} \log (\lambda_0 \theta \tau + 1)$$

where τ measures CPU-execution time, λ_0 is the initial failure intensity, and θ is a model parameter. Both these models have been used successfully in various telecommunication systems [40], and are often used together to bound the reliability predictions from above (basic Musa) and below (Musa-Okumoto).

Schneidewind Model

In Schneidewind's model [48], the expected number of failures m_i in the i-th period (all periods are of equal length) is given by

$$m_i = \frac{\alpha}{\beta}(e^{-\beta(i-1)} - e^{-\beta i})$$

with parameters α and β. This model is a special case of generalized Poisson model which allows the analyzer to choose different weights for the past observations to reflect the fact of closer linkage between reliability and recent past than that between reliability and distant past. Recently, Schneidewind also proposed a method to optimally select data to use with this model [49].

Littlewood-Varrell Model

In the Bayesian model by Littlewood and Varrell [32], the i-th inter-failure interval t_i follows the distribution:

$$f(t_i|\lambda_i) = \lambda_i e^{-\lambda_i t_i}$$

where the parameter λ_i follows a Γ distribution:

$$f(\lambda_i|\alpha, \psi(i)) = \frac{[\psi(i)]^\alpha \lambda_i^{\alpha-1} e^{-\psi(i)\lambda_i}}{\Gamma(\alpha)}$$

where $\psi(i)$ is an increasing function of i and α is a constant. Therefore, this model adjust the model parameter (λ_i) using the latest observations to analyze successive failure intervals (t_i). A wide variety of failure arrival patterns can be fitted to this model and analyzed accordingly.

Fitting and Using SRGMs

According to the specific types of data required for model fitting and usage, SRGMs described above can be grouped into time-between-failure (TBF) models or period-failure-count (PFC) models [24]. TBF models require data for successive inter-failure intervals, i.e., $\{t_i, i = 1, 2, \ldots, n\}$, where t_i is the observed time duration, measured in different time units discussed above, between the $i-1$-st failure and the i-th failure. PFC models require data for successive period lengths (t_i for the duration of the i-th period) and associated failure counts (f_i for the count of failures observed in the i-th period), i.e., $\{\langle t_i, f_i \rangle, i = 1, 2, \ldots, n\}$. Different models can be fitted to such data collected during testing with the help of certain software tools (Section 7). These fitted SRGMs can be used for various purposes, including:

- *Assessing current reliability.* The current reliability of a software system can be estimated as the fitted model function ($z(t)$, $\lambda(t)$, $R(t)$, etc.) evaluated at the current time.
- *Assessing reliability growth.* The overall reliability growth since the start of testing is captured by the fitted curve for the model, such as in Figure 8.
- *Predicting future reliability.* The reliability for any given time in the future can be predicted by extrapolating the fitted curves for corresponding models into the future.
- *Other predictions.* The time or resource needed to reach a given reliability target can also be estimated similarly by extrapolating the fitted curves, and calculating related quantities.
- *Exit criterion.* The modeling result can also be used to serve as an exit criterion for product release, i.e., the product can only be releases if it's estimated current reliability meet certain reliability target.

2.4 Input Domain Reliability Analysis

In the input domain reliability analysis, the reliability of a software system is defined to be the probability of failure-free operation for specific input states. Besides failure measurement, the key to reliability measurement in the input domain reliability modeling is input state measurement, which captures the information of precise input state for the software systems. This information can be related to testing results, and used for reliability modeling.

Input domain reliability models (IDRMs) generally use data from repeated random sampling to analyze product reliability. Therefore, they can be used directly for current reliability assessment and as an exit criterion for stop testing. Although IDRMs for a single set of data can not be directly used to analyze reliability growth, IDRMs for successive data sets over time or for subsets of data associated with certain input states can be used for various other purposes in reliability analyses and improvement. These novel usages of IDRMs will be explored further in Section 5.

Nelson Model

In Nelson's input domain reliability model [42], an unbiased estimation of reliability is the ratio between input states that result in successful executions over the total sampled input states. The unbiased reliability estimate $\hat{R}$ in the Nelson model can be derived from observations of running the software for a sample of n inputs according to the following setup:

- The n inputs are randomly selected from the set $\{E_i : i = 1, 2, \ldots, N\}$, where each E_i is a set of data values needed to make a run.
- Sampling probability is according to the probability vector $\{P_i : i = 1, 2, \ldots, N\}$, where P_i is the probability that E_i is sampled. This probability vector defines the operational profile.

- If the number of failures is f, then the estimated reliability $\hat{R}$ is:

$$\hat{R} = 1 - \frac{f}{n} = \frac{n-f}{n}$$

  That is, the estimated reliability $\hat{R}$ for a given input set equals to the number of successes over the total number of runs.

Notice that in the Nelson model, the operational profile and sampling probability distribution are handled implicitly.

Brown-Lipow Model

In the model proposed by Brown and Lipow [14], the whole input domain is partitioned into sub-domains; i.e., each E_i from input domain $\{E_i,\ i = 1, 2, \ldots, N\}$ represents a specific subdomain. The estimated reliability is:

$$\hat{R} = 1 - \sum_{j=1}^{N} \left(\frac{f_j}{n_j}\right) P(E_j)$$

where n_j is the number of runs sampled from subdomain E_j, f_j is the number of failures observed out of n_j runs, and $P(E_j)$ explicitly defines the probability that inputs in subdomain E_j are used in actual customer operational environment. This model adjusts for the different usage frequencies between the testing environment (as reflected by n_i as a proportion of all test runs) and the customer usage environment (as captured in $P(E_i)$), thus it is more widely applicable than the Nelson model. When there is an exact match between the two frequencies (i.e., $P(E_i) = n_i / \sum_{j=1}^{N} n_j$), the Brown-Lipow model reduces to the Nelson model.

IDRMs and Coverage Analysis

In both the Nelson model and the Brown-Lipow model, one common assumption is repeated random sampling without error fixing. However, in practical testing environments, whenever a failure is observed, appropriate actions are taken to identify, locate, and remove the underlying faults that have caused the failure. The reliability is changed due to defect removals. To assess the reliability at this point, another batch of runs needs to be executed, and the defect fixing problem arises again. A small subset of runs towards the end of testing can be used as a biased estimate of reliability. In general, the smaller the sampling window, the less bias there is. However, the confidence levels of the estimation are severely compromised because of the smaller sample size. This situation points to the need of using relevant time domain information to strengthen IDRMs, such as in the tree-based reliability models [55] using both time and input domain information discussed in Section 5.

Realizing the impracticality of failure detection without fixing, many researchers focus instead on maximizing the product coverage of test suites. The implicit assumptions here are twofold: 1) all detected defects will be removed, and 2) higher coverage leads to higher reliability. Consequently, the focus of this approach is not on the reliability assessment, but rather on increasing the various coverage measures that can be defined and gathered, and maximizing testing effectiveness defined accordingly [68, 72]. An alternative way of using coverage information in reliability modeling is outlined in [16], where coverage analysis results for individual test runs were used to weight time intervals based on the assumption that only test runs that cover new territories are more likely to trigger failures.

2.5 General Assumptions and Their Implications

There are various assumptions about the testing process, defect characteristics, distribution and handling assumed by the SRMs [24], including:

- *The software is used in an environment that resembles its actual usage by its target customers.* Operational profiles are commonly used to design, select, and guide the execution of test cases [39]. This assumption ensures the validity and appropriateness of reliability estimations and predictions.
- *Failure intervals are independent, which implies randomized testing.* This assumption restricts the usage of SRMs to late testing phases where randomized testing techniques are more likely to be used. While for earlier phases of testing, white box testing techniques are often used together with structural coverage based criteria for test process management [8].
- *Probability of failure in SRGMs is a function of only the number of faults existing in the software system, which implies an homogeneous distribution of faults.* Deviation to this assumption may lead to inaccuracies of SRGM results. Defect distribution information reflected in the input domain data can be used in conjunction with time domain information for reliability analysis and modeling. The integrated model in [55] is an example of recent development in this direction.
- *Time is used as the basis to define failure rate in SRGMs, which implies equivalence of time units.* This assumption places a stringent requirement for proper time measurement, making usage independent time measurements only suitable for systems with constant workload. For most software systems proper usage dependent time measurements need to be selected to capture workload variations for reliability modeling.

Many general issues about model limitations due to these assumptions are discussed in [24], and some particular observations relevant to large commercial software systems are presented in [60]. In the next section, we characterize a typical software testing environment for large software systems and examine the validity of the above assumptions and assess the applicability and effective of different types of SRMs.

3 Analyzing Reliability for Large Software Systems

Various software measurements need to be collected during testing for effective progress tracking, reliability assessment and improvement. However, before any such measurement and analysis devices are put into use in practical applications, a good understanding of the environment and the specific goals (or needs) of the measurement and analysis activities are necessary [5]. This section describes a typical testing environment, identifies specific needs and constraints for reliability analysis and improvement in large software systems.

3.1 Testing Process and Workload Characteristics

The products studied in this article are large software systems, where cost and time-to-market are some key concerns, competing with product quality goals. The product characteristics, market environment, and the software development and maintenance process all affect the suitability and effectiveness of various software technologies and analysis methods in project management and quality improvement.

Five large commercial software products, labeled A, B, C, D, and E, developed in the IBM Software Solutions Toronto Laboratory, were studied in a series of recent papers [33, 55, 56, 60, 62, 61, 64], and the results are summarized in this article. These products include relational database systems, language compilers, and computing environments. These products ranges from several hundred thousand lines of code to several million lines of code, and takes several years to develop, with a system testing usually lasting several months to over a year. The development process for these products [27] roughly conforms to the so-called "water-fall" model [71], with separate phases for product requirement analysis and specification, design, coding, and testing, although significant overlap and rework over succeeding phases are common. Component and system testing generally follow after coding and unit testing (which are done by the development team) and are performed by a dedicated testing team. The code base for each product under component and system testing is under continuous change because of defect fixing and the addition of minor functions. However, no addition of major functions is allowed in this phase, to ensure the product being completed and delivered

on schedule. Weekly status meetings are usually held to examine progress in terms of effort and quality improvement.

A commonly used testing strategy for such large software systems is to assure acceptable program behavior as defined by the product specifications through good coverage of all the major components and functionalities. High-level descriptions are used to describe test "scenarios". According to the different areas covered, these test scenarios can be grouped into different scenario types or classes. When testing is performed, a scenario description is used to construct an actual "test case", either manually or with the help of some test automation tools. Testing is also often divided among different testers and into several sub-phases, with a different focus for each one. This division allows the testing team to track progress on a smaller, more manageable scale.

When a defect is discovered by the testing team, a report is created and handed over to the development team. The testing team usually suspends activities related to the testing scenarios that have triggered the defect and continues with others. This approach reduces the chance of repeatedly finding duplicate defects that provide little additional information for defect removal or quality improvement. When an integrated fix for the reported defect arrives from the development team, the failing scenario is rerun and the testing process continues. However, these repeated scenarios do not generally lead to the repetition of exactly the same test cases.

Test workload usually varies considerably during the testing process of large software products. Testing activities and efforts are affected by several factors described below:

- *Shifting focuses.* Different testers usually concentrate on different functional areas and sub-phases. This division of testing activities forms some partial dependency sequences and causes test workload to fluctuate: 1) There is a learning curve associated with this shift that affects how much testing can be done; 2) As different parts of the products may differ in quality, the overall failure curve is also affected by the shifting focuses, depending on which part is currently under intensive testing. In addition, the quality of sub-parts within a focused area is more likely to be affected by defect detection and removal in the same area than from more distant areas. Therefore, such shifts can bring many "mini-steps" to reliability growth.

- *Progression of test cases.* For test efficiency concerns, more complex test cases are rarely attempted until most simple test cases have already been successfully executed. Simple test cases can be easily used to identify early (mostly easy to find) problems, while complex test cases can then be used to identify problems that have escaped earlier testing. Using complex test case early in testing may be problematic because it may be difficult to identify the large number of possibilities that may have caused the failure in the test run. This progression of test cases also has a direct effect on the progression of test workload over time.

- *Testing strategy.* A commonly used testing strategy, "*Test until it breaks*", also contributes to the uneven size of test runs and the progressively larger test runs. When this strategy is used, constant streams of user-oriented tasks are fed to the software system until it encounters a failure or until it reaches a pre-defined upper limit for test workload. This pre-defined upper limit is generally significantly higher than the "normal" workload for test cases. Therefore, the use of such testing strategies also contribute to the uneven distribution of workload during testing.

- *Staffing level variations* There is generally a gradual staffing up process in system testing. This factor, combined with the increased familiarity with the product as testing progresses, contributes in general to more (complex) test cases run towards the later part of testing. It is also common for a group of similar products to share a common testing team, and to have more than one product undergoing testing simultaneously. Sometimes, "emergency" situations may arise in some projects that require immediate response and re-allocation of resources, resulting in occasional wild fluctuations of staffing and workload levels for individual projects.

- *Code base stability.* Although major additions to the code base during component and system testing are not generally allowed, substantial change can be under way because of the detection of some major (high-severity or wide-spread) defects. This, in turn, affects the amount of testing that can be done and the number of potential defects (which may be masked or triggered by such changes) that can be detected.

Such workload variations need to be captured in the usage dependent time measurements to provide a solid data basis for reliability analysis and modeling. Some quantitative results in examining the workload variations in several such large software systems are presented in Section 4.

3.2 Specifying Testing Environment, Measurements and Constraints

A consistent view of the testing process and environment and a consistent interpretation of the collected data are important to ensure meaningful and appropriate analysis and modeling. To ensure such consistencies, we formally define the following terms:

Scenario: A testing *scenario* is a rough description of testing activities to be performed. It roughly corresponds to a collection of test cases. In contrast, the term *test case* in literature usually refers to a more precise description of testing activities to be performed.

Scenario type: Testing scenarios are grouped into scenario types or classes corresponding to the major functional areas they are designed to test. A scenario is usually identified by a name that consists of its scenario class name and a serial number within the class. Sometimes, multi-layered classifications are used to organized the large numbers of test scenarios for large software systems.

Run: An execution (also called an attempt) of a scenario is denoted as a *run.* For software systems that operate continuously, such as operating systems or telecommunication systems, a run is usually defined as a subdivision of the time period of program execution associated with user-oriented tasks [38]. Defect reporting is triggered by runs.

Failure: An unsuccessful run is treated as the arrival of a *failure.* An unsuccessful run is associated with the reporting of one or more valid defects.

Since a test scenario is only a high-level description of activities to be performed and components to be tested, and because the software under testing is continuously undergoing change because of defect discovery and fixing, each execution for the same test scenario will be different. A practical implication is that each run can be treated as an independent software usage operation.

All the activities for test tracking, analysis, and modeling require "good quality" (valid, consistent and useful) data. Numerous data are routinely captured in many projects for test tracking and testing process management purposes [56]. These data, with possibly minor modifications, provide the input for progress tracking and reliability analyses. The information associated with test runs can generally be grouped into three categories:

- *Result* of test runs. This includes result information regarding individual runs, summary information for a group of runs, and possibly defect classification information for related defects collected using some systematic scheme such as ODC (orthogonal defect classification [18]).

- *Timing*: This includes specific time associated with each run, such as start time, normal termination time for successful runs, or failure time for failed runs, and possibly software usage information during test runs. Such information is generally used for time domain reliability growth modeling.

- *Input*: The descriptive information about each specific run generally specifies the input for the program and the testing environment. Specific information may include scenario classification, tester, and possibly other input state identification, environmental setup, or specific input value.

In general, we can treat the data from testing as a multi-attribute data set with each data point corresponding to a specific run, and each data attributes corresponding to some specific information regarding the run. For input domain analysis, we need to analyze and establish relations between input information and test result. For time domain reliability modeling, we need to analyze timing information and test result data.

To implement an effective strategy for test tracking and reliability analysis and to introduce process and quality improvement that can last, we have to overcome a number of obstacles within the constraints of project environment. We have to work within the project schedule and minimize the risk of project delays and cost overruns because of the introduction of measurement and analysis activities. We also need to address various organizational issues to ensure effective and long-lasting improvement and process changes.

The uncertainty or risk associated with introducing a new technology and the unfamiliarity with the measurements and analysis methods can have a major effect on the success of such measurement and improvement programs. Adequate lead time should be allocated for planning, education, and training, to overcome inertia and dispel misunderstandings. An evolutionary approach need to be adopted, to demonstrate benefit step by step, eventually leading to a full implementation of a comprehensive measurement and analysis strategy. A gradual change is also more likely to be ingrained into the corporate culture and is more likely to last.

Some organizational issues also need to be addressed. In most large software development organizations, the product (or project) organizations, including software developers, testers, and project managers, are responsible for producing various software products to meet market needs; while a separate quality organization oversees the product quality and development process across all the different products. The product organizations are generally vertically integrated according to product lines and market segmentation. The quality organizations are generally structured horizontally, with each quality analyst or team covering a number of software technologies (such as design methods, code complexity analysis, testing tools, etc.) or process areas (e.g., process definition and modeling, defect prevention). Involvement, buy-in, commitment, and cooperation of the developers, testers, managers, and quality analysts are essential to the success of any measurement and improvement initiatives. A successful strategy for test tracking and analysis generally requires close collaboration between the product and quality organizations.

3.3 Reliability Analysis in Scenario-Based Testing

A quality measurement that is close to the customer view need to be used to track and optimize testing activities. To satisfy this need, software reliability and related measures can be used to assess the overall quality of a software product by fitting various SRGMs to observed data. On the other hand, because product quality is generally uneven across different components or functions, there is a strong need to identify those parts with substantially more defects, so that greatest improvement can be made with the least cost. Unfortunately, traditional SRGMs provide little such information. Analyses are needed to help in risk identification and management. In this way, various management decisions, such as resource allocation, test case selection and execution, can focus on those high-risk areas. Such information can also help us to plan for product service and for future update releases.

Overall, the mixture of structured (centered around the framework of scenario classes), clustered (focused on fault localization), and randomized testing, rather than purely randomized testing, dominates in large software system testing. Test workload also varies considerably, and defect distribution are non-uniform. All these seem to deviation from the common assumptions of various software reliability models (SRMs) summarized in Section 2.5. We next examine these issues to assess the applicability of existing SRMs and the need for new models and analysis methods:

- *Approximating scenario-based testing with random testing.* Despite the individual dependencies due to structured testing according to scenarios and scenario classes, testing is generally conducted by different testers in parallel, interleaving in some arbitrary fashion. As a result, the overall testing resembles random testing. The key to this approximation is parallelism and interleaving. If substantial

part of the test runs strictly follows some sequential order, such as stress testing follows after other testing scenarios in [60], test runs and related data need to be divided into subsets (or sub-phases) with SRMs fitted to individual subsets. On the other hand, if fundamental differences exist among different scenario classes, they should be analyzed separately, such as in [55] for different sub-products.

- *Dealing with defect fixing effect.* Each discovery of a defect generally triggers related test runs to locate the defect and additional test runs to verify the defect fix. However, in large software systems, usually hundreds or even thousands of defects are discovered and fixed during testing [55, 60]. When such large numbers of defects are discovered and fixed, the overall testing still resembles random testing, because of the lack of longer term dependencies among different testing periods despite local dependencies. Therefore, when using SRMs, we have to make sure that adequate failure data exist for statistical modeling.

- *Selecting appropriate time measurement to capture test workload variations.* Because test workload usually varies consideration for large software systems, proper usage dependent time measurements, including test run count, execution time, and transactions need to be selected to capture workload variations for reliability analysis using SRGMs [60, 62].

- *Dealing with uneven distribution of faults.* Faults are generally distributed quite unevenly across components, with new and changed components and those with high complexity usually containing significantly more faults than the rest [66]. The uneven distribution of faults implies non-homogeneous chances of defect detections or failure arrivals when different components or subsets of operations are tested (only a portion of faults is at risk of being detected). This kind of uneven distribution can be characterized by various input domain reliability models with explicit usage of input states and testing results. Tree-based reliability models (TBRMs) can be used to systematically handle input domain partitions, and help identify high risk areas (high likelihood for triggering failures) and guide focused remedial actions [55]. A general consequence of using such an approach is that the product quality across different components or functional areas will even out towards the end of testing, by then SRGMs become more applicable because the closer conformance to the assumption of homogeneous chances of fault detections.

To summarize, various SRMs are applicable under the scenario-based testing environment for large software systems. However, care must be taken to ensure certain conditions being met for effective use of measurement data to track progress and SRGMs to assess and predict reliability. A new reliability model, TBRM [55], can be used to analyze both time and input domain information for risk identification and reliability improvement. All these analysis and modeling activities, together with their integration and tool support, will be discussed in the rest of the article.

4 Usage Measurement and Reliability Analysis with SRGMs

In this section, we examine two important data entities for large software system in their system testing: testing activities and defects, and presents reliability analysis results using time domain software reliability growth models (SRGMs) for several large software products. These results are presented in two ways:

- *Graphical* presentation to allow for visual inspection to detect trends and patterns between fitted models and actual observations. Either the cumulative data and models (e.g., the left graph in Figure 8) or the rate data and models (e.g., the right graph in Figure 8) can be used.

- *Tabular and textual* forms to present important model parameters and summary statistics, such as in Table 4 and related discussions. Sometimes, this information can also be directly presented within graphs, such as in Figure 9.

4.1 Test Workload Measurement and Reliability Growth Visualization

One common assumption in SRGMs is elapse time captures test workload and can be used as the basis to define failure rate and reliability (Section 2.5). This assumption implies equivalence of time units and places a stringent requirement for time measurement for reliability modeling. Usage independent time can be measured by various physical measurement of time of different granularity, ranging from rough calendar dates to precise time stamps. All usage dependent time measurements can be associated with test runs. Because the test cases are generally grouped into scenario classes according to customer usage information, each test run during testing represents a well defined unit of software usage linked to some user-oriented operations. Consequently, various test workload measurements based on runs can serve as the basis of time measurement for reliability modeling:

- *Test run count.* The numbers of test runs associated with segments in the overall test execution sequence can be used as time measurements [60].
- *Execution time.* For runs falling into a given period, the time they actually spent in execution can be tallied for reliability modeling. This alternative was proposed by Musa [40], and has been successfully used in various software systems, particularly in telecommunication systems and some operating systems.
- *Transactions.* In this alternative, detailed task measurements for test runs, generically referred to as transactions, can be used as time measurement for reliability modeling [62].

Notice that in both execution time and transaction measurements, the measurement activities have to be carried out dynamically because of the possibility for failures during test runs, and the in-feasibility of predicting actual execution time or transactions before execution. On the other hand, test runs can be measured more easily: We only need to indicate whether a run takes place in the overall sequence of runs. The cost for data collection in the above three alternatives also varies considerably: Test runs can be captured easily and with minimal cost; Execution time measurement involves the use of some independent monitor of system usage; Transaction measurement involves the use of specific tools that not only monitor system usage but also the specific types of usage to capture transactions. The collection of these measurements and related software tool support are discussed in Section 7. Each of the above time measurement alternatives as well as calendar time workload characterization will be examined in detail below.

Tracking Workload and Failures over Calendar Time

To examine the workload progression and distribution, data from test activity logs were extracted for products A, B and C [60] and plotted in Figure 1. The sequences of bars from left to right represent the daily workload from the beginning to the end of system testing. Each bar indicates the number of testing runs for a particular day, with scales shown along the corresponding vertical axes.

It is immediately obvious from Figure 1 that the daily workload distributions are quite uneven, with significantly more runs on certain clusters of days than others. We can also see a general trend of intensified testing effort as testing progresses, shown by the taller bars at later part of testing (towards the right in each plot). Other products (D and E) we studied also demonstrate similar characteristics [55, 62]. These workload characteristics are affected by a number of factors discussed in Section 3.1. Figure 1 provides a quantitative characterization of their effect on workload characteristics.

Other entities, such as failure count and workload measured in execution time or transactions, can be measured and track in similar ways. However, to compare the overall progress and trend, cumulative plots, such as in Figure 2, are often more informative because several entities can be presented side-by-side to allow for comparative examination. One major concern in such plots is the simplicity of the graphical representation. Therefore, for general tracking purpose, only essential information needs to be included, with the x-axis representing calendar time and the y-axis representing workload or defect count to plot 1) progression of workload (cumulative runs, execution time, and transactions) over time, and 2) progression

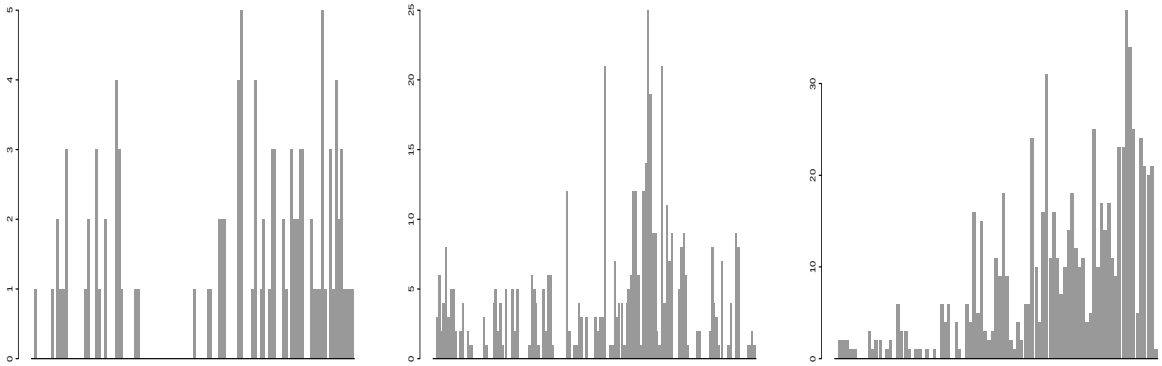


Figure 1: Daily test runs for products A (left), B (middle), and C (right)

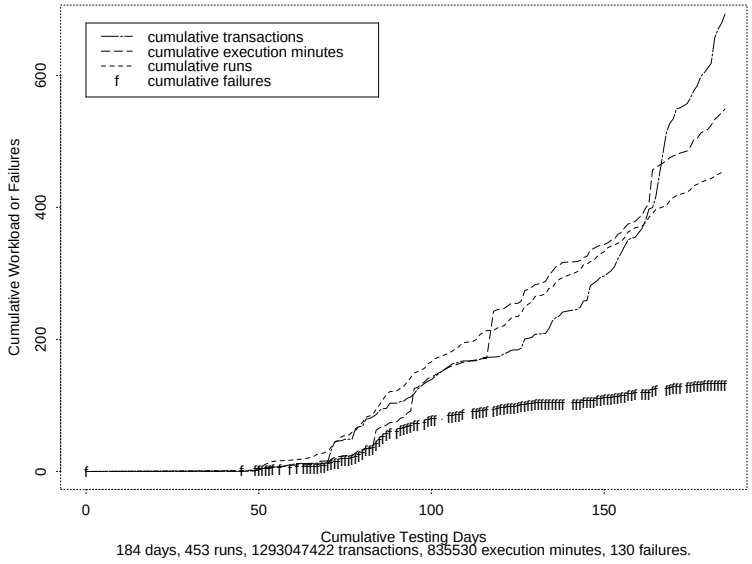


Figure 2: Testing activity and defect tracking (product E)

of cumulative of failures over time. Summary information about total runs, failures, and defects to date can also be included for easy reference.

Figure 2 tracks the test effort and failure observations for product E studied in [62], with the summary statistics explicitly shown below the x-axis. The curves for cumulative runs and failures are shown to scale, while the curves for cumulative execution time and transactions are using different scales not explicitly shown (but can be deduced from the summary statistics shown below the x-axis). The information presented in graphs such as Figure 2 can be used consistently over the whole testing process, helping product organizations to visualize general effort and defect trends and variations, and manage project resources and schedules.

Visualizing Reliability Growth

Data visualization presents data in visual and graphical forms. It is a powerful tool that preserves much of the information and structure in the data, and provides a good way to examine the data for trends, distributions, and is particularly good for identifying anomalies [20]. As a result, graphical representation of data and analysis results is used throughout this article. In particular, failure arrivals over time can be visualized and examined for overall trends and patterns. Reliability growth over time can be characterized

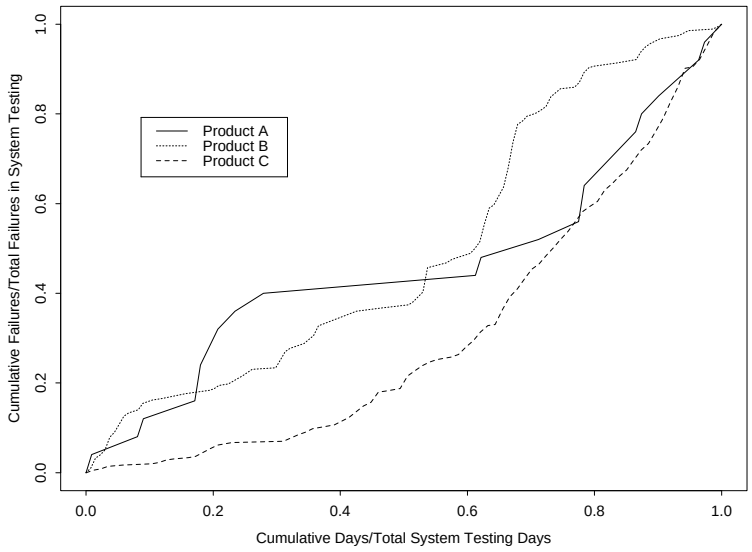


Figure 3: Failure arrivals in calendar time for products A, B, and C

by the generally longer and longer between-failure intervals as time progresses. When the cumulative failures (vertical axis) is plotted against time (horizontal axis), such as in Figure 2, the effect of reliability growth can be visualized by the leveling-off tails in the failure arrival curves, indicating gradually fewer failure arrivals towards the end. As seen from Figure 2, there is no obvious trend of reliability growth over calendar time (see the cumulative failure curve) for product E, and no reliability growth models can be fitted to these data. The failure curve largely follows the general trend of workload (cumulative runs) curve. As expected, the number of runs in a given period has a strong influence on how many failures are expected for that period.

Figure 3 shows the cumulative failure arrival curves, in calendar time, for the whole system testing phase for products A, B, and C. To better compare trends and patterns, normalized scales are used in Figure 3. The horizontal axis is the ratio of cumulative testing days to total system test days, and the vertical axis is the ratio of cumulative failures to total failures observed in system testing. A quick inspection of the curves reveals no trend of reliability growth over calendar time exemplified by the leveling-off effect with time progression.

Usage Dependent Time Measurement

Test runs and execution time can be defined and measured independent of the specific product under study by counting the test runs and monitoring CPU or system usage (Section 7). However, transactions are product specific workload measures and need to be defined and measured accordingly. Recently, transaction measurement and reliability modeling were performed for product E [62], a relational database management system product (RDBMS), and is used below to illustrate the definition and measurement of transactions for reliability analysis.

A transaction in RDBMS products can be defined as an operation on a row in any table in the database manager. Operation in this context consists of either the updating, selection, insertion or deletion of a row. In effect the transaction count for an application program or a test run is the sum of the number of rows updated, selected, inserted or deleted by the application or corresponding test case when they are executed. This definition differs from commonly used definition of database transaction, where it corresponds to a sequence of operations which transforms a database from one consistent state to another consistent state [21]. This conventional definition still covers transactions whose workload varies greatly, thus it is not suitable as a detailed workload measure.

Figure 4 presents the transaction measurement result for product E [62]. The sequence of vertical lines

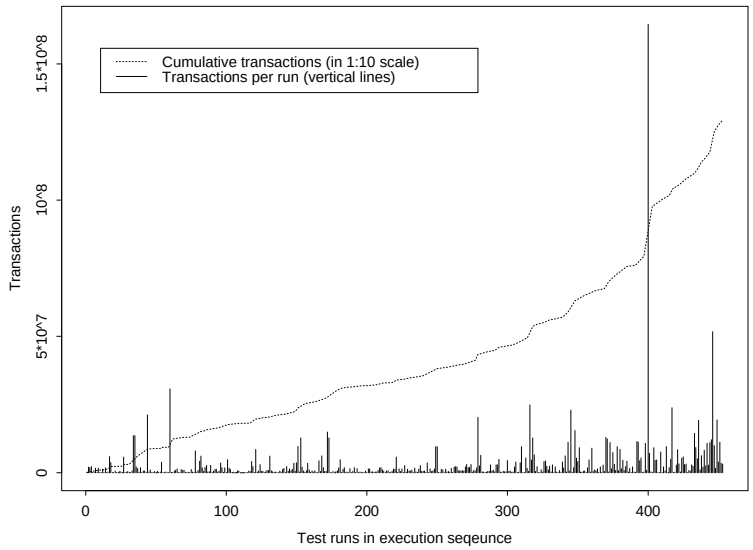


Figure 4: Measured transactions (per run and cumulative, product E)

represents the transactions for individual test runs presented in their execution sequence. The height of each vertical line represents the transactions for a particular run, with the scale shown by the vertical axis. The dotted line shows the cumulative transactions over the entire sequence of test runs, shown in a reduced 1:10 scale to highlight the effect of individual transactions on the cumulative transactions. As clearly visible in Figure 4, the distribution of transactions per run is quite uneven, and there is a general trend of intensified workload as testing progresses. This agrees with earlier observations comparing execution time to test runs [60]. This transaction measurement result gives us a quantitative characterization of the uneven workload distribution and the intensified workload as testing progresses.

Comparing Reliability Growth in Different Time Measurements

Figure 5 plots the failure arrivals in product E against cumulative transactions, and contrasts it with failure arrivals indexed by cumulative test runs and days. To better compare the shapes and patterns, common scale of 0 to 1 for the horizontal axis (time axis) is used, representing the ratios between the cumulative days, runs, or transactions to that of total days, runs, or transactions respectively used in the plot in Figure 5. The general failure arrival patterns are examined below:

- *Failure arrivals in calendar time*: There is no pronounced reliability growth in calendar time because of the uneven workload distribution. In fact, the failure arrivals do not seem to follow any specific pattern except in the very beginning, where a slow start of failure detection is common.
- *Failure arrivals in test runs*: The pattern is more consistent than calendar time indexed failure arrivals. For the initial portion lasting until approximately 60% to 70% of total test runs, a seemingly concave curve is visible with gradual flattening. However, after that, the failure arrivals seem to follow a steeper curve again. Figure 4 offers at least a partial explanation to this observed pattern: the cumulative test workload as measured in transactions follows a somewhat linear curve before steep upturn at around 60% to 70% of test runs. Consequently, the intensified test runs as testing progresses induces bias in the pattern of failure arrivals against test runs, particularly late in testing.
- *Failure arrivals in transactions*: Failure arrivals in transactions demonstrate considerable reliability growth, changing from steep slope in the beginning to flattened slope towards the end. This pattern also provides further evidence to the effectiveness of using integrated analysis technique [55] for risk

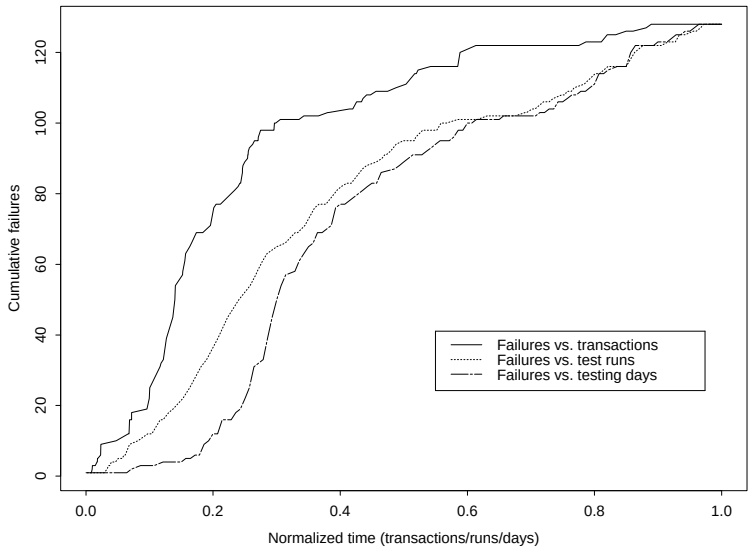


Figure 5: Failure arrival patterns in different time (product E)

identification and reliability improvement early in testing (to be discussed in Section 5). It provides us with a better time measurement for reliability modeling.

4.2 Using Run Count and Execution Time Failure Data in SRGMs

As discussed in Section 3 and quantitatively examined above, test effort as measured by workload per time period varies considerably for large software systems in general. Therefore, calendar time based measurement data are not suitable for reliability analysis using SRGMs. Consequently, in [60], an earlier study of products A, B, and C, only test run count and execution time were used to index failures for reliability modeling.

Model Fitting and Overall Observations

Figure 6 plots the actual failure arrivals against runs and execution time for product B, and shows all the fitted models individually. Several of the SRGMs summarized in Table 2 were attempted, including BMB, BMP, GO, JM, LV, Geo, Musa, MO, and S. Notice that all the fitted models for each data set (failures against runs or execution time) give almost identical fitted curves, appearing to collapse into a single curve in Figure 6. Similar patterns were observed for product A and C [60], and so are conclusions in the subsequent discussions. Notice that the total number of failures plotted against execution time is less than that plotted against runs. The reason is that runs that consume no execution time were omitted in execution time normalization of the data. Most reliability models assume that no more than one failure can occur simultaneously. To be consistent with this assumption, all runs with zero execution time were omitted in execution time modeling.

Comparing to failure arrivals in calendar-time (Figure 3), the normalization effect of runs and execution time is immediately visible in Figure 6. A general trend of reliability growth over time is visible in all the failure arrivals against runs and execution time. Furthermore, the actual failure arrival patterns seem to conform to the fitted models fairly well, which is in sharp contrast to the seemingly random failure arrivals against calendar time in Figure 3.

The curves of failure arrivals plotted against runs seem to be more stable both over time and across different products. This stability in failure arrival patterns leads to more consistent modeling results from different reliability growth models. SRGMs can also be fitted to these kind of data fairly early in system testing, making these data and models good devices for test and reliability progress tracking. However, it is

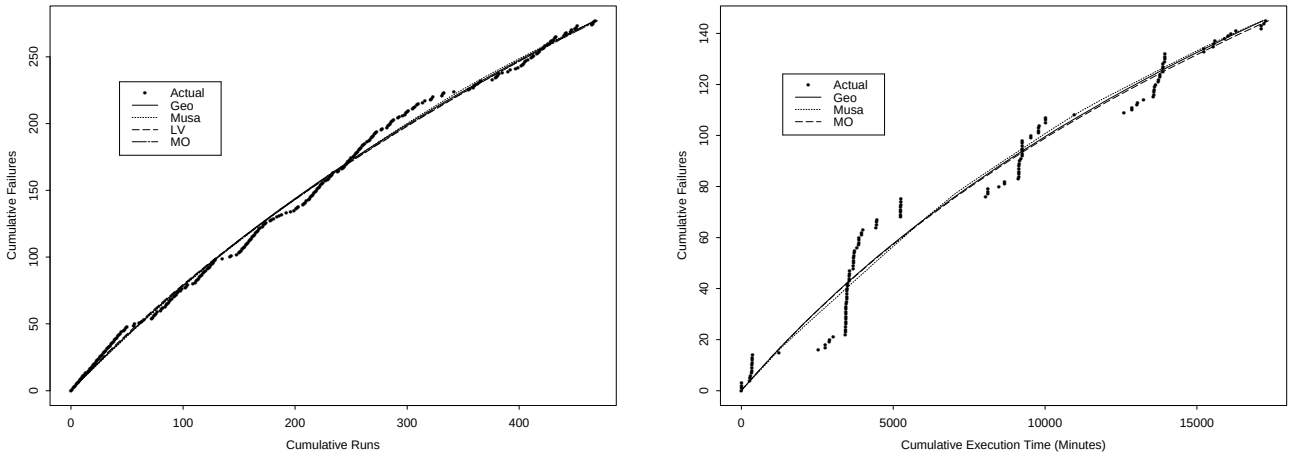


Figure 6: SRGMs with run count (left) and execution time (right) data for product B

hard to assure the equivalence of runs in terms of to what degree each run exercises the software. As testing progresses, the test cases generally also become more complex and exercise the software more intensively. This fact causes biases in reliability estimations using such run based models.

Although execution time seems to provide more information about test case executions and testing effort, it does not always provide a better picture of reliability growth. Because of the nature of the products A, B, and C, many complicated test scenarios involve little CPU execution time, thus making CPU execution time not always the preferred choice for time measurement used to normalize failure intervals. Setups and manual operations represent a major portion of many runs, but represent very little chance of encountering failures because they involve mostly the repeated usage of tested-and-proven operations. Such non-homogeneity of test scenarios have a strong effect on the probability of encountering a failure and therefore on the measured reliability.

Because of the problem with CPU execution time normalization, another variation, system execution time was also tried. System execution time for each run is the difference in time-stamp values between start and termination of the run. In this case, the problem with setups and manual operations is magnified because the low CPU utilization in such cases is not accounted for, resulting in no improvement over the CPU execution time normalization. In fact, some deterioration in model fit was observed.

As seen in Figure 6, sudden change and clustering of failure intervals measured in execution time dominate the overall failure arrival patterns. SRGMs can only be fitted to failure interval data normalized by CPU execution time well past the 50th percentile of system testing. Further more, the arrival of new clusters and gaps can easily make predictions of earlier fitted models grossly inaccurate, and lead to quite different parameter estimations and reliability assessments. As a result, such models are highly unstable, and sensitive to the arrivals of new failure data. For system execution time normalization, this problem is even more severe than for CPU execution time normalization. The end result of all these is the inappropriateness of execution time based models for reliability tracking for these products, especially in the early phases of system testing.

Modeling for Scenario Groups and Testing Sub-phases

To track the reliability growth and assess product reliability with respect to certain groups of operations represented by different test scenario classes, separate reliability models can be constructed. Typical scenario classes used in testing product B are organized into 12 categories by the system test team. These 12 scenario classes can be collapsed into four related scenario groups. Of the runs falling into the four groups, group 2 has too few runs to warrant reliability growth modeling, and group 4 consists mostly of runs without CPU execution (zero CPU execution time). As a result, SRGMs were fitted to run based failure interval data for

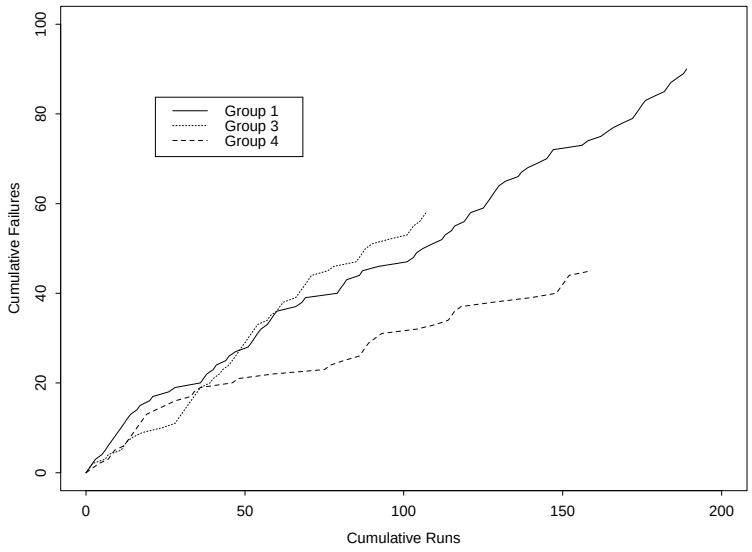


Figure 7: Failure arrivals vs. runs for 4 scenario groups (Product B)

groups 1, 3, and 4, and to execution time based failure interval data for groups 1 and 3. This approach is similar to the approach in [45], where SRGMs were fitted to data associated with different types of defects.

Figure 7 shows the cumulative failure curves against cumulative runs for all test cases selected from groups 1, 3, and 4 respectively. The reliability growth models were fitted to failure interval data in runs for each group individually. Similar to the overall models above, all such run based models fit well with actual observed failures. Reliability assessments from different models for a given set of data are almost identical. However, for group 4, considerably more reliability growth is realized than for the other two groups. This can be clearly visualized in the failure arrivals plotted against runs for the three groups in Figure 7: After an initial period of similar failure arrival rates (failures per run), the failure arrival curve for group 4 levels off considerably more than the curves for the other groups. The current estimated reliabilities for these three groups are quite different, which points to the need for focused testing of certain features or functional areas, — a potential explored further in our tree-based reliability models (TBRMs) in Section 5.

Group 1 consists of mostly communication related functions, where extensive and continues usage are the norm. Execution time based SRGMs for group 1 fit much better to actual failure observations, compared to the same models for overall data in Figure 6. This result indicates the applicability of execution time based models to communication type of operations. On the other hand, no model can be fitted to the execution time failure data for group 3, where I/O related operations dominate.

Lessons Learned in Modeling Reliability for Products A, B, and C

Despite the problems of estimation biases, run based models are robust across data sets and can always provide at least a rough estimate of the reliability of the products under testing. Therefore, they should always be used as a starting point. If CPU execution time based failure data reflects the main testing activities and operations, models could be fitted to such data to get more accurate results. In addition, inter-group differences between the run based and execution time based models seem to dominate intra-group variations for these products. As a consequence, the selection of data normalization (what time unit to use) should be the primary concern in practical applications. In many cases, multiple normalizations could be done to offer different but complementary views of reliability.

In general, the product characteristics and testing environment have a strong influence on the observed failures and measured reliability. As seen from the modeling results for the four scenario groups and individual testing sub-phases, information about restricted subsets of data could be used effectively to build SRGMs

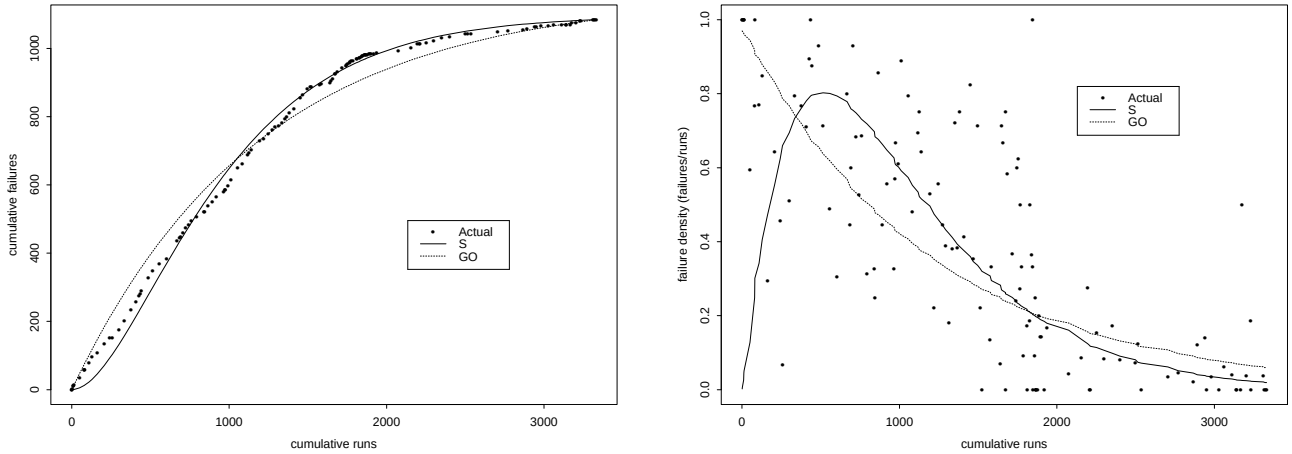


Figure 8: Reliability modeling with SRGMs for product D

to better assess reliability. When CPU execution constitutes the dominant feature of operations, such as in communication related scenarios and in the stress-testing sub-phase, execution time based models should be selected to give better reliability assessment.

SRGM results for subsets of data can also be used to guide testing activities. For example, with Figure 7 and related modeling results showing that product B is less likely to experience scenario group 4 related failures, the testing team can now shift their attention and focus their effort away from such scenarios. In general, testing can be focused on those "bottleneck" components or system functions which are more likely to experience failure than others. New test cases can be designed, enhanced, and executed with this focus in mind. In this way, the overall reliability target can be achieved under minimal cost.

The overall conclusion from studying the reliability of products A, B, and C [60] is that SRGMs are applicable to large software systems under scenario-based system testing, and can provide fairly accurate measurements and predictions of product reliability, provided that the time measurement reflects system usage or workload. This earlier study also identified various issues that were studied in followup studies [55, 62], including general guidelines for selecting proper time measurement, the need for alternative time measurement under specific situations, and the usage of data partitioning information for risk identification and reliability improvement.

4.3 Using Homogeneous Test Runs for Reliability Modeling in Product D

As demonstrated in the previous study [60] summarized above, the key to good modeling result using test run count as time measurement is to ensure homogeneity of test runs. In product D studied in [55], a large amount of such small, homogeneous test cases were used, making test run count reflects test workload and activities and therefore a good usage dependent time measure.

Figure 8 plots the cumulative failure data (left graph) and the failure intensity data (or rate data, right graph) indexed by cumulative runs for product D, and two reliability growth models (GO and S) fitted to these data. Besides GO and S models, other models from Table 2 used to fit these data include JM, SW, GP, BMB, and BMP. JM, GP, BMB, and BMP models fit the observed data well, and follow curves very similar to that of GO; therefore, they are omitted in the plot to avoid overcrowding. Because all of these models essentially agree with each other on their estimations, an overall consistent quality assessment can be formed with a relatively high degree of confidence. The S-shaped model, which is relatively optimistic, is used to bound these estimations from above. On the other hand, SW model is discarded because it does not fit the observed data well.

Table 4 summarizes some important information for all the fitted models for product D using the data of period failure count with period length measured by the number of runs. To examine the progression of

data set	cumulative days	runs	failures
F_3	126	1847	979
F_2	152	2444	1026
F_1	160	2898	1063
F	177	3331	1084

Table 3: Data used for reliability analysis (product D)

Model	Data Set F_3			Data Set F_2			Data Set F_1			Data Set F		
	MTBF	$\hat{N}$	SSQ	MTBF	$\hat{N}$	SSQ	MTBF	$\hat{N}$	SSQ	MTBF	$\hat{N}$	SSQ
JM	2.86	1857	168	6.8	1230	263	11.2	1174	308	17.7	1152	323
SW	7.69	2902	669	23.1	1097	699	51.3	1077	825	60.2	1094	793
GP	1.89	1751	165	4.1	1240	240	6.4	1204	270	11.8	1167	323
GO	2.78	1881	168	6.6	1233	260	10.8	1178	294	16.9	1154	312
BMB	2.78	1870	167	6.7	1231	259	11.0	1176	303	17.2	1153	319
BMP	2.86	1847	167	6.7	1229	259	10.9	1177	302	17.2	1153	317
S	4.76	1112	412	14.7	1065	416	28.5	1083	444	52.8	1095	462

Table 4: Reliability modeling results (product D)

models over testing process, models were fitted to four forward inclusive sets of data identified as F_3, F_2, F_1 and F. Data set F corresponds to the entire system testing and the data used in Figure 8. The cumulative data summary for each of these data sets is given in Table 3. The modeling results summarized in Table 4 include:

- Estimated mean-time-between-failures (MTBF), which captures the overall product reliability in an easy-to-understand summary measure. Larger MTBF indicates higher reliability.
- Estimated total defects ($\hat{N}$) by different models. The difference between $\hat{N}$ and the current cumulative defects (number of failures observed so far) gives us an estimate of the number of defects still remaining in the system. This information can be used in practice to plan for product support and customer service after product release.
- Sum of squares for the residuals between model prediction and actual observations, SSQ. SSQ is a goodness-of-fit statistic, reflecting the degree of how satisfactorily the models fit actual observations. Smaller SSQ indicate better fit, and increased confidence in the assessments and predictions made by the models.

As we can see from Table 4, the modeling results by different models are converging, giving us stronger confidence in the modeling results as testing progresses towards the end. Future predictions can be easily derived from these models, picking specific points corresponding to management and scheduling concerns. For example, one major concern in project management is to predict the estimated reliability at the scheduled product release date. For product D, predictions of product reliability at product release (corresponding to data set F) made based on earlier data (data sets F_3, F_2, F_1) conform well with assessment made with the use of final data set F. These converging results and good predictions confirm the appropriateness of assessing reliability with the failure count data normalized by runs for this product. Therefore, SRGMs fitted to failure data indexed by test runs can provide accurate measurement and prediction of reliability and other quantities of concern (e.g., resource and schedule to reach a reliability goal), if homogeneity of test runs can be assured.

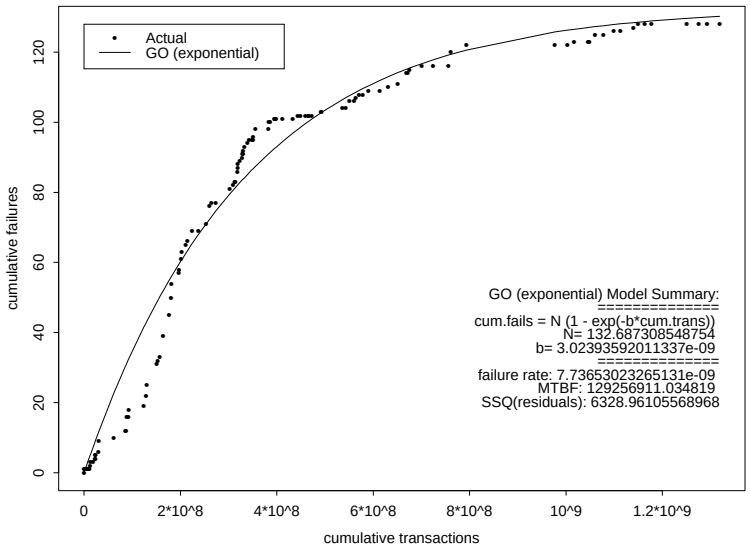


Figure 9: Transaction based modeling result (product E)

4.4 Transaction Measurement and Reliability Modeling for Product E

Intrinsically diverse test cases often need to be executed for some large commercial software systems, such as product E, a workstation based relational database product (RDBMS) developed in the IBM Software Solutions Toronto Laboratory, with a large user population and diverse usage environments. Different users might want to use different features of the product in different environments. For example, some users might use product E as a stand alone RDBMS, while others might use it as a server serving a large number of networked clients. To ensure correct operation under such diverse usage environments, test cases of different characteristics and vastly different complexity have to be constructed.

The usage of such diverse test cases makes it inappropriate to use raw test runs as time-index for failure data in reliability modeling. In addition, it is often hard to break down the larger test cases into smaller ones because of the tight coupling among different subtasks and execution subsequences. Under such situations, neither the homogeneity of individual test runs be assured nor can they be broken down to homogeneous sub-runs. Other alternative finer-grain measurements, such as transactions defined in Section 4.1, need to be gathered.

Figure 9 plots the failure arrivals against cumulative transactions and the fitted Goel-Okumoto model (GO) [25]. Various other models listed in Table 2 were also tried on the failure arrival data indexed by transactions. However, as we observed in Section 4.2, individual model variations for a given set of data provide mostly similar results, falling into a narrow band. Consequently, only the modeling results from Goel-Okumoto model were presented, with the discussions equally valid for results from other models.

Figure 9 gives the model formula for the Goel-Okumoto model ($m(t) = N(1 - e^{-bt})$ from Section 2.3, with `cum.fails` in place of $m(t)$; and `cum.trans` in place of t), the fitted model parameters N and b, as well as modeling result summary including: 1) *current failure rate*, failure rate function $\lambda(t) = m'(t)$ in the NHPP model evaluated at the current time, 2) the estimated meantime between failures *MTBF*, and 3) sum of squares of the residuals between fitted model and actual observations SSQ. A visual inspection of the failure arrival data and the fitted model confirms the good fit between the two, and the reasonable assessment given by the model. This is particularly true when this is compared to the models fitted to test runs and execution time for products A, B, and C. SRGMs were fitted to testing data on a weekly basis near the end of testing. Modeling results for these forward inclusive data sets (similar to that in Table 4 for product D) were examined and the following observations can be made about these modeling results (details in [62]):

data set	cumulative measurement			model parameters		modeling results	
	runs	trans ($\times 10^6$)	failures	N	$b \times 10^{-9}$	MTBF$\times 10^6$	SSQ
F	455	1318.68	128	132.7	3.02	129.3	6329
F^1	470	1506.98	128	131.7	3.07	244.5	6375
F^2	472	1507.10	128	131.6	3.07	254.2	6381

Table 5: Re-modeling with new data (product E)

- *Consistency*: The models fit consistently well with the observed failure arrivals over transactions as compared to models for similar products (products A, B, and C studied in [60]) where test runs and execution time were used as time measurements. This confirms the appropriateness of using transactions as the test workload measurement for product E.
- *Convergence*: The models seem to converge as time progresses. The estimated model parameters seem to converge to a very narrow band, which give us more confidence in the assessment and prediction results given by these models.

Testing continued after product release but with more strict control to prepare for the next refresh for product E. Because of this, two additional sets of data, identified as F^1 and F^2 at approximately one week and one month after F respectively, from after this release point F were obtained to evaluate the applicability and effectiveness of SRGMs fitted to transaction indexed failure data. However, since only a low number (close to zero) of failures are expected at this stage, we compare the model at F with models at F^1 and F^2, the cumulative data set with all the new data rolled into existing data.

In general, remodeling can be done for each new set, appending newly accumulated data to the existing data set to form a forward inclusive series of data sets. The modeling results, with new data rolled in, are summarize in Table 5, showing data set characteristics (cumulative runs, transactions and failures), fitted model parameters (N and b), as well as modeling results (MTBF and SSQ). The same observations as given above for the series of models before the cutoff are also valid in these models: the high degree of conformance between models and actual observations, and the stability of fitted models, with estimated model parameters falling into an even narrower band. These results further confirms the applicability and effectiveness of using transactions for usage measurement and reliability analysis for product E. In general, property defined and captured transaction measurement can provide a solid basis for accurate reliability measurement and analyses using SRGMs fitted to transaction indexed failure data.

4.5 Effective Usage of SRGMs in Large Software Systems

As demonstrated in this section, software reliability growth models (SRGMs) can be used effectively to measure and model the overall quality of large commercial software systems, provided that proper test workload measurements are captured for effective reliability modeling. Some key findings from the earlier study of products A, B, and C [60] described in Section 4.2 include:

- *Robustness of run count as time measurement.* This time measurement provides good consistency and traceability to track testing progress. Models based on such data can give good reliability assessments if runs are homogeneous across test cases and over time.
- *Limited applicability of execution time models.* Execution time based models provide better reliability measurement than run based models for specific subgroups where continuous and intensive usage is the norm. For the overall systems, because of the non-homogeneity of testing activities with some involving little execution and much setup and manual operations, such models are not as robust as run based models.

Not much can be done about the limited applicability of execution time, because the characteristics of operations are determined by the customer usage information and product functionality. However, test runs

can be organized into different structures and executed in different sequences. Therefore, these early results can be extended to derive conditions for selecting proper test workload measurements:

- *Use of homogeneous runs and condition.* Because of the low cost and robustness, test runs can be used as time-index to failures for reliability modeling, if homogeneity of test runs can be ensured, or if larger test runs can be broken down to smaller, homogeneous sub-runs. These conditions provide constructive information for future applications.
- *Use of transactions as detailed workload measurement.* When the above conditions (homogeneity and divisibility of runs) can not be met, transaction measurement can be used to provide data that are not only usable for reliability modeling, but also meaningful to customers so that modeling results can be easily used for project management to ensure quality from a customer perspective. Reliability modeling results using transaction-indexed failures confirm the appropriateness of transactions as the usage dependent time measurement.

There are many research issues for more effective use of SRGMs in software reliability assessment and prediction: a small set of general rules can be developed to define appropriate transactions for different software systems, possibly with automated support for measurement capturing. Another way to correct for the estimation biases in the SRGMs using coarse grain test runs for time measurement is to calibrate data input or modeling output. This calibration is dependent on adequate historical data and consistent trend in the measurement bias that can be quantitatively characterized. For example, if the test runs grow in size in a linear form as test progresses, this linear relation could be used to weight the test runs so that the failure intervals measured in the weighted runs more accurately reflect software usage. The key to this result calibration is the existence of adequate measurement data. These data can also be used to establish functional relations between test case progression and biases in reliability modeling results to calibrate the modeling results.

5 Tree-Based Reliability Models (TBRMs)

SRGMs offer overall reliability assessments and predictions for software products, but provide little information on how to improve reliability. However, there is a strong empirical evidence that defect distribution is quite uneven in software products, hence there is a strong need for risk identification and management [11, 50, 66]. Alternative analysis techniques and models are necessary in order to identify and correct problems for effective reliability improvement.

In an earlier study [60] summarized in Section 4.2, we observed that measuring and modeling reliability for different sub-groups of test scenarios or testing sub-phases provide valuable information about product reliability from a different perspective. Such input domain information can be used systematically in conjunction with time domain and failure information traditionally used in SRGMs for problem identification and reliability improvement in a recently developed approach [55] using tree-based models. We next describe these tree-based reliability models (TBRMs) and discuss their applicability and effectiveness in reliability analysis and improvement.

5.1 Integrated Analysis and Tree-Based Modeling

In the input domain reliability analysis using IDRMs (Section 2.4), there is a precise definition of input states and usage of such information in reliability modeling. The reliability of a software system in IDRMs is defined as the probability of failure-free operation for a set of input states randomly sampled according to its operational profile [24, 54]. IDRMs can be easily extended to model reliability for data subsets. Each subset of data can be defined by some specific input state sub-domains where the inputs are sampled from or specific time periods when the inputs are sampled. This kind of approach could provide up-to-date assessment of reliability if only proper subsets of recent runs are selected. Partitions based on the input states and the

0. *Initialization.* Create a list of data sets to be partitioned, referred to as `Slist`, and put the complete data set as the singleton element in `Slist`. Select the size and homogeneity thresholds T_s and T_h for the algorithm.

1. *Overall control.* Remove a data set S from `Slist` and execute step 2. Repeat this step until `Slist` becomes empty.

2. *Size test.* If $|S| < T_s$, stop; otherwise, execute steps 3 through 6. $|S|$ is the size (or the number of data points) of data set S.

3. *Defining binary partitions.* A binary partition of S is a division of data into two subsets using a *split condition* defined on a specific predictor p by a comparison operator "<" and a cutoff value c. Data points with $p < c$ form one subset (S_1) and those with $p \geq c$ form another subset (S_2). If p is a categorical variable, then an unique binary grouping of its categories forms a binary partition.

4. *Computing predicted responses and prediction deviances for S, S_1 and S_2.* The predicted value $v(S)$ for the response for a set S is the average over the set; i.e., $v(S) = \frac{1}{|S|}\sum_{i \in S}(v_i)$; and the prediction deviance $D(S)$ is $D(S) = \sum_{i \in S}(v_i - v(S))^2$, where v_i is the response value for data point i.

5. *Selecting the optimal partition.* Among all the possible partitions (all predictors with all associated cutoffs or binary groupings), the one that minimizes the deviance of the partitioned subsets is selected; i.e., the partition with minimized $D(S_1) + D(S_2)$ is selected.

6. *Homogeneity test:* Stop if this partitioning cannot improve prediction accuracy beyond a threshold T_h, i.e., stop if $\left(1 - \frac{D(S_1)+D(S_2)}{D(S)}\right) \leq T_h$; otherwise, append S_1 and S_2 to *Slist*.

Table 6: Algorithm for tree-based model (TBM) constructions

models built for these partitions can be used to analyze reliability variations across different functional areas or product components, and to identify problematic areas. Such problematic areas can be characterized by the low reliability for the partitioned subsets and the conditions for these partitions. However, there are a large number of possible partitions, which can not be handled adequately by *ad hoc* partitions such as in [60]. Systematic ways of handling the partitions are needed.

As stated before in Section 3.2, the information associated with test runs includes test results, input state and environmental information, and timing information. We can associate each run with a data point, and treat these individual pieces of information as individual data attributes. Among these attributes, some are numerical (timing and workload, failure count, etc.), while others are categorical (scenario classification, environmental identification, defect type, etc.). To integrate the analysis in a systematic way, we need the support of analysis techniques and tools that can handle this diverse information. Tree-based modeling technique [19, 69] and related software tools (Section 7) are selected to support such integrated reliability analyses.

Tree-based modeling (TBM) is a statistical analysis technique that attempts to establish predictive relations through recursive partitioning. This technique originates from the social sciences, where the data often consist of both numerical and non-numerical components with complicated interactions, making classical statistical models inadequate [19]. Selby and Porter pioneered the use of this technique to analyze various software engineering data [44, 50]. We have also used this technique to analyze defects and metrics data for various software systems recently [57, 63, 66], and gained valuable experience in handling such data and interpreting analysis results for large software systems.

In tree-based models (TBMs), modeling results are represented in tree structures. Each node in a tree

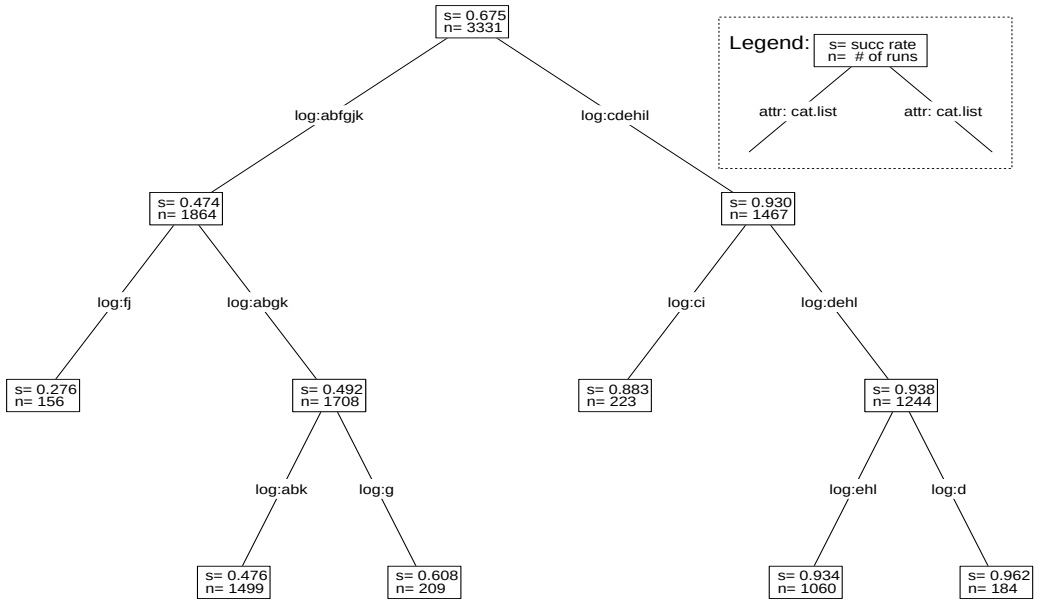


Figure 10: Reliability across input sub-domains *log* (product D)

represents a set of data, which is recursively partitioned into smaller subsets. The data used in such models consist of multiple attributes, with one attribute identified as the *response* variable (or response) and several other attributes identified as *predictor* variables (or predictors). Recursive partitioning minimizes the difference between predicted response values and the observed response values. The process of recursive partitioning is computationally intensive, thus proper software tools are needed to support its implementation. A binary variant of recursive partitioning is used, with the specific tree construction algorithm summarized in Table 6. Tool support for TBM and other analyses is discussed in Section 7.

5.2 Reliability Analysis for Partitioned Subsets

With each run treated as a data point, Nelson [42] estimate of reliability for any given subset of runs can be easily derived. Let n_i be the number of runs falling into a specific subset i, and f_i be the number of failures. The estimated reliability $\hat{R}_i$ for subset i according to the Nelson model is: $\hat{R}_i = 1 - f_i/n_i$ (Section 2.4). Each subset can be defined by some partition conditions on input sub-domains which need to be satisfied by all the runs falling into it, with the specific partitions selected by the recursive partitioning algorithm (Table 6). Each input subdomain generally corresponds to some function areas or operations.

Figure 10 is a TBM that provides reliability estimates for different input sub-domains. Each subset of data associated with a node in the tree is uniquely described by the path from the root to that node, with each edge along the path representing a split condition. To label the nodes, we use an "l" to indicate a left branching, and an "r" to indicate a right branching, following the sequence from the root to the specific nodes. For example, for the tree in Figure 10, the third leaf node from left, with 209 runs and 0.608 success rate, is labeled "lrr".

The success rate s_i in Figure 10 for the subset of runs associated with a tree node i is the Nelson reliability $\hat{R}_i$. The split conditions, written at the edge leading to each node, define the groupings of input sub-domains. In this example, each input subdomain is denoted by its test log (data attribute *log*) where the data is taken from. From the TBM in Figure 10, we can see that the input sub-domains defined by runs from test logs f and j (156 runs associated with node "ll", with success rate of 0.276) have particularly lower reliability than other sub-domains, a signal of possible problems that need to be addressed.

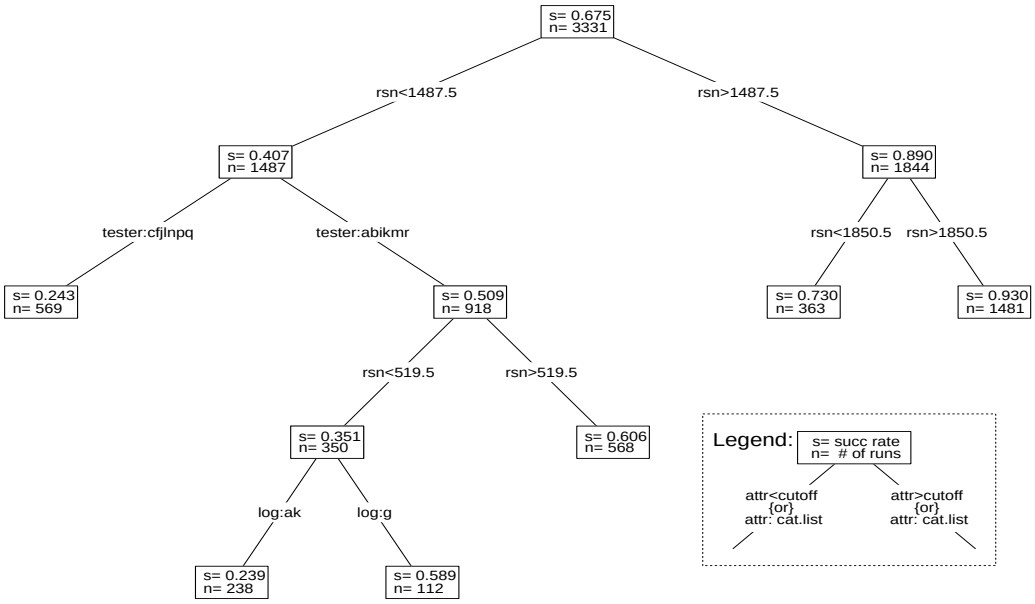


Figure 11: Tree-based reliability model (TBRM) for product D

For a subset of runs falling into a time period, the estimated failure rate and MTBF can be obtained from the Nelson model, by viewing the serialized runs as equivalent repeated random sampling. This can be viewed as a special case to the above input domain reliability model based on partitions. When draw graphically, such as in Figure 14, this time partition based model gives us an piecewise linear curve for the cumulative failures, maintaining constant failure arrival rate for each time segment. The interpretation and usage of such piecewise linear SRGMs are discussed in Section 6.

5.3 Analyses Integration and TBRM Applications

For tree-based reliability models (TBRMs) that integrates both time and input domain analysis of reliability, the result indicator can be treated as the response variable, and the timing and input state information associated with each run as the predictor variables. The result indicator r_{ij} (the j-th run from subset i) has value 1 for a successful run or 0 for a failed run. Let there be n_i runs with f_i failures for subset i. The predicted result s_i for subset i according to Step 4 of the algorithm in Table 6 is:

$$s_i = \frac{1}{n_i}\sum_{j=1}^{n_i} r_{ij} = \frac{n_i - f_i}{n_i} = \hat{R}_i$$

which is exactly the Nelson estimate of reliability $\hat{R}_i$. This result can also be easily mapped to the estimated failure rate $\hat{\lambda}_i$ or MTBF $\hat{T}_i$ in the time domain. As a result, the TBRM for the combined data give us a systematic way of analyzing reliability for the partitioned subsets, providing reliability estimates that have valid interpretations in both the time domain and the input domain.

Figure 11 shows a TBRM that uses both the time domain attributes and the input domain attributes in reliability modeling. Among the multiple time and input domain attributes from Table 7 supplied as predictors, only a subset is selected by the tree construction algorithm (Table 6) to recursively partition the data. The selected partitions and associated split conditions are local optimums according to the selection criterion (Step 5 in the algorithm of Table 6) for TBM construction. Therefore, the ordering of split conditions from the root to the leaves of the tree represents the progression from more general and important characteristics for the analyzed data to more specific ones. For example, for the tree in Figure 11, *rsn* cutoff

- Timing: calendar date (*year, month, day*), *tday* (cumulative testing days since the start of testing), and *rsn* (run sequence number, which uniquely identifies a run in the overall execution sequence).
- Input state: *SC* (scenario class), *SN* (scenario number), *log* (corresponding to a sub-product with a separate test log) and *tester*.
- Result: *result* indicator of the test run, with 1 indicating success and 0 indicating failure.

Table 7: Data attributes (in *italic*) in TBRMs for product D

at 1487 represents the most significant turning point in reliability growth that also dominates reliability variations across different input sub-domains. For a general understanding of important characteristics that differentiate data into different levels of reliability, only those few high level partitions near the root need to be considered. These high level partitions identify key factors linked to reliability.

For a set of data associated with an interior node of a tree, if a time predictor is selected by the algorithm in Table 6 to partition the data set, we can interpret that reliability change over time is predominant. The binary partition distinguishes two clusters of runs, one of higher estimated reliability after (or before) certain cutoff time and another of lower reliability before (or after) that cutoff. If an input state variable is selected as the predictor to partition the data set, the interpretation is that the product is more reliable in handling certain subsets of input states than others, and this input state partition is linked more closely to reliability differences than partitions by any other factors. What is more, the split condition characterizes the two partitioned subsets of the input states. This information can be used to guide further analysis to find out if indeed certain functional areas or components are of lower quality, and if so, appropriate remedial actions can be carried out.

The main usages of TBRMs include:

- *Assessing reliability and factors closely linked to reliability.* Each node in the tree gives the estimated reliability for a specific subset, and the series of split conditions from the tree root to individual tree nodes give us a good picture of important factors linked to product reliability.
- *Assessing current reliability and predicting future reliability.* When timing variables are selected as predictors in tree based models, the partition that corresponds to the latest time segment can be used to assess current reliability and to predict future reliability. This subject is examined further in Section 6.
- *Monitoring change in reliability over time and across different input states.* As testing progresses, new trees can be constructed to analyze the constant streams of new data. These progressing trees provide us with a picture of changing reliability and changing key factors linked to reliability. Consequently, such a series of trees help us track the progress in testing and reliability.
- *Identifying problematic areas for further analyses and remedial actions.* Subsets of runs with exceptionally low reliability can be easily identified in TBRMs and can be characterized by the split conditions leading to the nodes associated with those subsets. The low reliability indicates possible problems in some related functional areas or product components. Further analyses, such as root cause analysis, can be performed for these subsets to identify the problems and devise remedial actions. Since the weakest part of a product determines to a large degree the overall reliability, the identification of such problematic areas also represents great opportunities for cost effective reliability improvements. The partitioning information, obtained from the path leading to those specific low reliability subsets of runs from the root of the tree, can be used to guide causal analysis and help devise remedial actions.
- *Enhancing existing exit criteria.* A product should exit from testing only if its latest TBRMs show no particularly low reliability subsets of runs that can be traced to specific functional areas or product

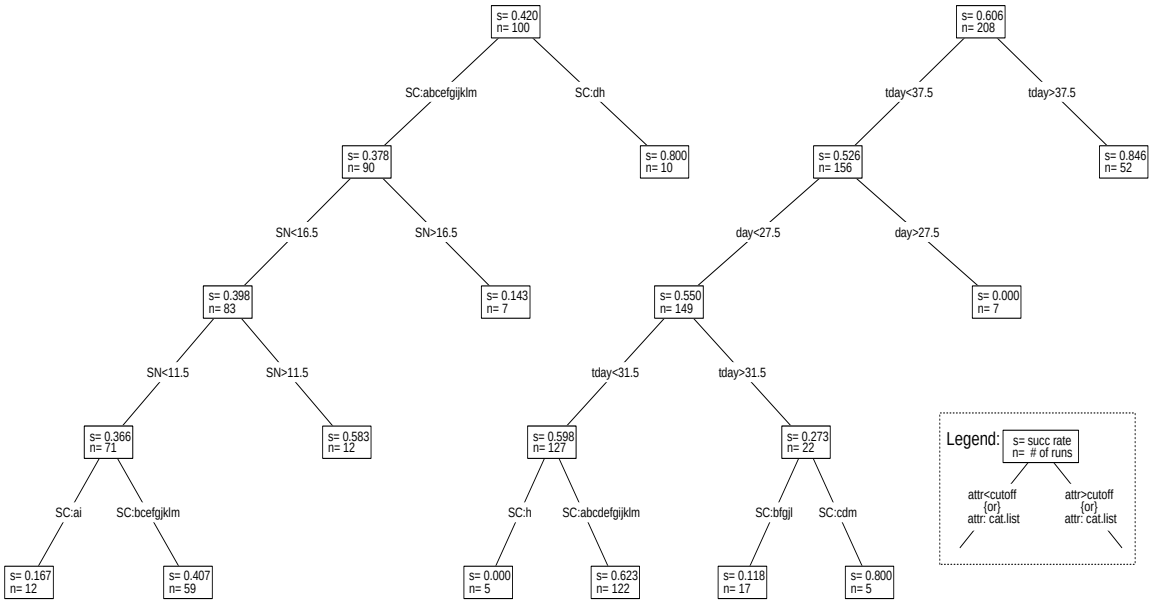


Figure 12: Tree-based reliability modeling for a sub-product in product D

components. These trees at the testing exit point can be typified by major partitions, particularly those associated to runs from later portion of testing, defined on timing variables, with subsets associated with later runs demonstrating higher reliability than earlier subsets. The TBRM in Figure 11 is such an example.

To summarize, TBRM provides two key benefits: 1) a systematic way to analyze software reliability using both input domain and time domain data through data partitioning, and 2) information for problem identification and reliability improvement in addition to reliability assessment and prediction. These benefits and most of the above usages cannot be achieved individually by SRGMs nor by IDRMs.

5.4 Experience Using TBRM in Product D

Tree-based reliability models (TBRMs) has been applied to several products developed in the IBM Software Solutions Toronto Laboratory, including products D and E studied in [55, 62]. The data attributes for product D used in the previous and subsequent sample TBRMs in this section are listed in Table 7. Among the nine data attributes, *SC* and *tester* are **categorical** variables and the rest are **numerical** variables. The result indicator, *result*, is used as the **response** variable; and the rest are treated as **predictor** variables. The complete date information, or its equivalent *tday*, can be used to indicate the overall time passage. However, there might be some seasonal fluctuations that affect the measured reliability. To assess such seasonal effects, the individual date variables (*year, month, day*) were also included in the modeling.

The left tree in Figure 12, covering data at approximately the halfway point of testing for a sub-product in product D, represents typical results in the earlier part of testing. Here, the most important factor linked to the estimated reliability is the test scenario information (*SC* and *SN*), which indicates that different parts or function groups are of different quality. Notice that scenario number *SN* also reflects test case characteristics because most obvious scenarios were usually constructed first (low *SN*) to cover major functions in the scenario class for product D, and then "odd" scenarios were constructed later (high *SN*) to complete the coverage. From this tree, we can see that the estimated reliability is 0.8 for runs selected from scenario classes d and h (node "r"), a significant difference from runs from other scenario classes (node "l"). We can also identify some subsets with very low success rates: nodes "llll" (the leftmost node) and "lr" (the

center node with $s = 0.143 \wedge n = 7$). With the help of further analysis that identified other characteristics of the runs associated with these nodes, appropriate remedial actions were taken to detect and remove related defects.

The right tree in Figure 12, covering the whole testing phase for the same sub-product, represents typical results towards the end of each testing phase. The time information has become a major factor linked to success rate. Worth noting is the primary split at the root, with test runs in the later days ($tday > 37.5$) having much higher success rate (0.846) than early ones (0.526 for $tday < 37.5$). Reliability growth over time dominates reliability variations across different functional areas or product components. All the subsets associated with particularly low success rates (nodes "llll", "llrl", and "lr") are associated with early runs. This kind of result can be interpreted as the product becoming more homogeneous in quality as time progresses, as reflected by similar success rate across different test scenarios and major function groups for later runs.

For the products we studied, such progression is universal. The presence of this latter kind of trees was used as an exit criterion for product release. Throughout testing, a series of conscious decisions was made to focus on problematic areas, which led to a product exiting the testing with uniformly high quality. This is in sharp contrast to products A, B, and C, where tree-based models were only used as a post-mortem analysis [57]. In those earlier products, the quality of different parts were still quite uneven as they exited from testing.

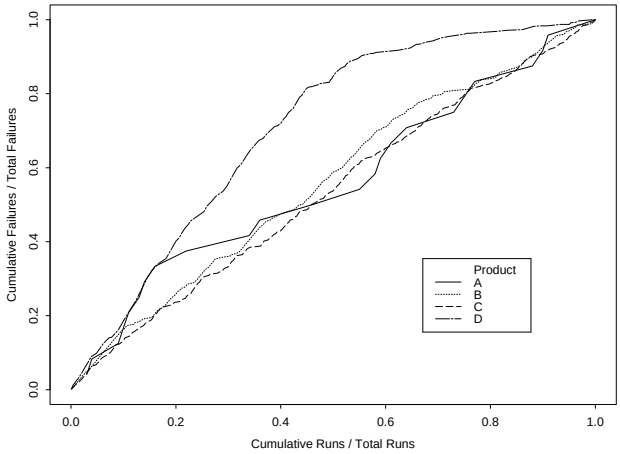
The same kind of analyses as described above were carried out for each sub-product or sub-product group. When the sub-products were integrated for the bundled release, integration testing was performed. Figure 11 shows the TBRM built for the complete set of data for product D covering all the components and the whole testing. For this specific tree, the data set for node "ll" has much lower reliability than the rest of the tree and could be analyzed further. However, as we can see from Figure 11, these are the runs executed early in testing ($rsn < 1487.5$), and further analysis showed that the problem has been corrected. Therefore, this product satisfied this exit criterion, and was fit for release to the marketplace.

5.5 TBRMs' Impact on Reliability Improvement: A Cross Validation

TBRM analysis results were actively used by managers, developers and testers to focus on problematic areas for reliability improvement, leading to products exiting testing with uniformly high quality. A shift of focus from reliability assessment to reliability improvement is evident, particularly in the early part of testing. This shift of focus also helps the closer collaboration between the testers who detect failures and the developers who fix defects, because more potentially useful information is presented in the tree structures. TBRM results are more likely to be used by the developers to fix discovered defects and to focus on highly problematic functional areas or product components.

TBRM's effectiveness in improving reliability can be examined visually by looking at it's effect on failure arrival curves. Trends and patterns in the failure arrival data for product D can be visually examined and compared to three earlier products, A, B, and C studied in [60] and summarized in Section 4.2, where the TBRMs were not used. These products share many common characteristics that make them comparable: They are similar in size, and all developed in the IBM Software Solutions Toronto Laboratory using similar development and testing processes. Figure 13 plots the failure arrivals against cumulative test run counts for products A, B, C, and D. Normalized scales (cumulative runs or failures as a proportion of total number of runs or failures, respectively) are used to better compare the failure arrival patterns: Visibly more reliability growth was realized for product D than for the other products as seen by the more concave curvature and much more flat tail. This is a good indication that TBRM helped improve the reliability for product D, particularly because early problematic areas were identified and effective remedial actions were carried out.

To compare the reliability growth quantitatively, *purification level* ρ as defined in [55] can be used. ρ captures overall reliability growth and testing effectiveness, and is defined to be the ratio between the failure rate reduction during testing divided by the initial failure rate at the beginning of testing. Notice that ρ is unitless and normalized to have the range spanning between 0 and 1. This definition avoids the problems in comparing such measures as failure rate or MTBF, which may not be valid because the unit used (in this



Purification Level ρ	Product			
	A	B	C	D
maximum	0.715	0.527	0.542	0.990
median	0.653	0.525	0.447	0.940
minimum	0.578	0.520	0.351	0.939

Where: $\rho = \frac{\lambda_0 - \lambda_T}{\lambda_0} = 1 - \frac{\lambda_T}{\lambda_0}$

λ_0: failure rate at start of testing

λ_T: failure rate at end of testing

Figure 13: Comparison of failure arrivals for products A, B, C, and D

case *runs*) may not be equivalent across different products.

The table in Figure 13 shows the purification levels estimated by different fitted SRGMs, with the maximum, minimum, and median estimates for ρ shown, for the four products. In product D, where TBRMs were actively used, there is a much stronger reliability growth as captured in the purification level ρ than in the earlier products. This quantitative comparison strongly support the effectiveness claim of TBRMs. In fact, both the TBRM and SRGM results were used to support the decision to release this product ahead of schedule. And preliminary in-field data also seemed to support this early release decision, with the observed in-field defects at or below the projected level.

5.6 Findings and Future Development of TBRMs

The tree-based reliability models (TBRMs) built on both time domain and input domain measurement data combine some strengths of the existing SRGMs and IDRMs, and can help us improve the reliability of software products as well as assessing them. Initial results from applying this approach in several software products developed in the IBM Software Solutions Toronto Laboratory demonstrated its effectiveness and efficiency. Some key findings are summarized below:

- *Integration*: Combining the ability to provide overall reliability assessment by the time domain reliability growth models and the ability to associate product reliability to subsets of input states in input domain reliability analysis, TBRM offers an integrated framework that provides both reliability assessment and guidance for reliability improvement.
- *Applications*: TBRM provides models that identify key predictors to the differences in the measured reliability. Problematic areas can be easily identified and characterized, and remedial actions focused on those areas can be derived. The analysis results can also be used to manage the testing process and to enhance existing exit criteria by evaluating reliability levels and homogeneity across functional areas or product components.
- *Efficiency*: TBRM provides efficient utilization of the measurement data collected for general test tracking purposes, providing all the above benefits at little additional cost.
- *Cross validation*: Initial results comparing products where TBRM was used versus earlier products demonstrated the effectiveness of this approach in reliability improvement.

1. Determine the period P.
2. Identify each test run (run i) falling into the period ($i \in P$).
3. Count the number of runs, n, and the number of failures, f, for the period P.
4. Compute the transactions for the period, t, as the summation of the transactions for all the individual runs, t_i for the i-th run, falling into the period. That is, $t = \sum_{i \in P} t_i$.
5. Compute the failure rate, λ, as failures divided by either runs or transactions, i.e., $\lambda = f/n$ or $\lambda = f/t$.

Table 8: Generic procedure for period failure rate computation

Future research includes alternative analysis techniques that also use other information collected during software development process for process and quality improvement. Primary candidate data for these analyses include early measurement data, such as design and code metrics data for different products [15, 66], and detailed defect data collected under some systematic framework such as ODC (orthogonal defect classification [18]). The selection of appropriate measures to use in such expanded reliability analyses can be guided by formal models for measure evaluations and selection [65]. In terms of the analysis technique, alternative tree structures, other structured analyses, and synthesized analyses can be explored. For example, natural groupings and hierarchies of input states can be used directly to form multi-way partitions instead of binary partitions. Graph structured analysis method, such as optimal set reduction [12], can also be used. For data associated with a tree node, SRGMs can be used instead of the Nelson model. This synthesis of tree-based models and reliability growth models would be very computationally intensive and requires the selection of appropriate algorithms and good software support.

The strategy described in this section tends to even out reliability variations across different subsets of data associated with different operations or components. However, operational reliability for the software in the field depends on the actual operational profile [39]. Consequently, such information can be used in TBRMs so that we can focus on the operational profile weighted reliability instead of raw reliability for different subsets. This way, we can identify areas that have the most impact on operational reliability and focus reliability improvement actions on them.

6 SRGM Based on Data Clusters (SRGM-DC)

In practical applications, it is common to have considerable data fluctuations, such as observed in Section 4.1 when we examined the test workload measurement results for several large software systems. Such fluctuations are usually caused by variations in product and process characteristics. As discussed in Section 3, in the scenario-based testing of large software systems, there may exist dependencies among test runs within a short time-window, but there is little long term dependencies, and the overall testing process still resembles random testing at a coarse granularity. Different data clusters may have different failure rates, with little dependency between clusters. Properly grouped data, such as using clusters with similar failure rates, can generally reduce such fluctuations and produce models that conform more closely with the observations. Tree-based models (TBMs) can be used to derive a new type of software reliability growth models based on these clusters, and filter data for use in traditional SRGMs [61]. We next discuss this new modeling approach.

For a given time period, the average failure rate, referred to as the *period failure rate*, can be defined by the number of failures per unit time to approximate the instantaneous hazard function defined in Section 2.3. The generic computational procedure for this period failure rate is outlined in Table 8. The number of failures can be easily counted, and the time period length can be measured using different time measurements. As

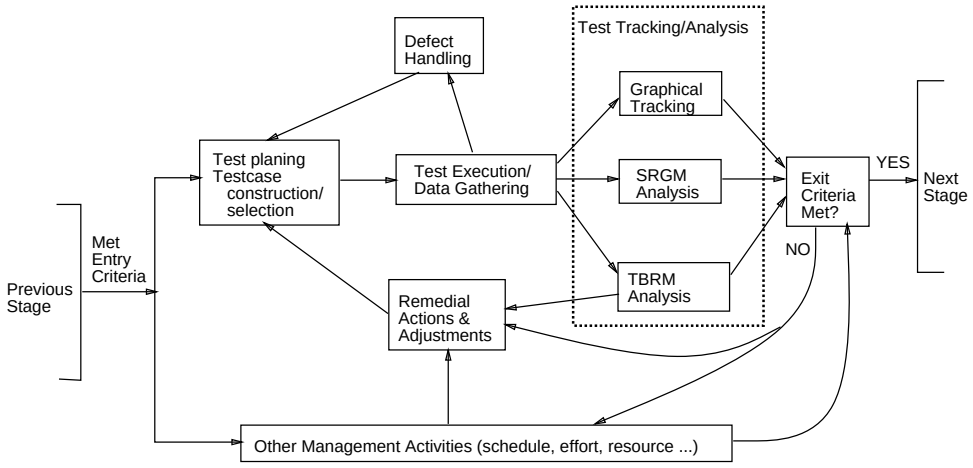


Figure 18: Integrating Tracking and Analysis into Testing Process

for n_i, and the ratio between the ith transactions over the total transactions $(n_i / \sum_i n_i)$ for p_i, then the SRGM-DC for the segment is identical to the Brown-Lipow model.

Similar to the Jelinski-Moranda model [28] and Moranda's geometric model [36] (JM and Geo models) summarized in Section 2.3, where failure rate remains constant during inter-failure intervals, constant failure rate is assumed in SRGM-DC for each time period within a data cluster identified by TBMs. When the failure rate plot is drawn, such as in the rate plot (right) in Figure 16, the failure rates over segments form a step function similar to JM and Geo models, but with different step sizes (both up and down) determined by the observations in the segment instead of down-steps of constant step-size (JM) or geometrically diminishing step-size (Geo). SRGM-DC do not assume a functional form for failure rates over different time segments associated with data clusters. SRGM-DC is data driven and data determined, depending on the exact clusters of data in the input. This is similar to the approach in the Littlewood-Verrall model [32] also summarized in Section 2.3, where the failure rate varies with latest observations.

There are many research issues: A pilot study applying this model in an integrated fashion with other reliability measurement and improvement initiatives is planned. A thorough study comparing different data grouping techniques could also help us identify better rules for data grouping for specific applications. Progress in these research initiatives could yield a set of alternative modeling techniques to traditional SRGMs that can be more effectively used for reliability assessment as well as readily integrated with TBRMs for reliability improvement.

7 Integration, Implementation, and Tool Support

The analysis and modeling activities described in the previous sections share many common sub-activities, and need to be implemented under the environmental constraints and supported by selected software tools. We next summarize relevant research in this area [56, 64] and discuss issues about activity integration, practical implementation, and software tool support.

7.1 General Implementation and Process Linkage

When the analysis and modeling activities described in the previous sections are carried out in the testing process, they represents changes to the existing testing process, as illustrated in Figure 18. These activities can be grouped into the following three categories in the modified testing process (shown within the dotted rectangle in Figure 18):

- *Graphical tracking of testing efforts and defect trends.* Various data visualization techniques are used to examine the general trends and patterns of efforts and failure arrivals, and to visualize testing progress and reliability growth over time.
- *Reliability analysis and modeling.* Various existing SRGMs and SRGM based on data clusters (SRGM-DC) can be fitted to observed testing data to assess and predict product reliability and help make product release decisions.
- *Tree-based reliability modeling (TBRM) for problem identification and risk management.* This includes analyzing the complete set of project testing data by constructing TBRMs, identifying problematic areas, and guiding process adjustment and remedial actions.

This combination of analysis and modeling activities attempts to utilize the collected data to the fullest potential. SRGMs are mainly used to evaluated product "readiness" and serves as an exit criterion, while TBRMs are central to guiding remedial actions. Over time, these two analyses also complement one another nicely: Early in testing, local data fluctuations tend to dominant the overall data pattern, and the trend of reliability growth is not expected to be very visible. The focus of analysis should be on using TBRMs to identify early problematic areas and guide improvement actions. Toward the end of testing, as a result of actions focused on problematic areas early on, there should not be any particularly weak part in the software system. The focus of analysis is now shifted to using SRGMs to assess overall product quality and to evaluate product "readiness" for release.

Analysis, modeling, and result presentation require good understanding of the models and analysis techniques and familiarity with various analysis and modeling tools. Thus it requires the involvement of some dedicated quality analysts to work in cooperation with the product development organization, possibly over a long period of time for effective implementation and technology transfer. As mentioned before in Section 3, in most large software development organizations, the project organizations are usually organized around product groups, while the separate quality organizations are usually organized around software technology areas. Long term collaboration between the two organizations are required for effective implementation and technology transfer:

- At the beginning of the engagement, the responsibility of the project organization is limited to ensuring accurate, timely, and consistent data collection, and using the analysis results provided by the quality analyst to manage their projects.
- As the engagement proceeds, the product organization will gain understanding and familiarity with the analyses and related tools, and could become more effective in implementing such analyses themselves because of their deep product and process specific knowledge. The longer-term goal is to package the experience, much like in a mature experience factory [4, 6], and transfer the analysis methodology and make long-lasting process improvements.

The project organizations eventually take over implementation of the strategy for test tracking and analysis, performing most or all of the analyses themselves.

7.2 Tool Support for Data Collection, Analyses, and Presentation

The individual activities described above share many common elements: Data must be collected; appropriate analysis and presentation techniques must be selected to handle the diverse information. Various software tools are needed to support these common activities:

1. *Gathering testing measurements.* Defect information, as well as time and input domain testing measurements, need to be gathered using various tools to provide input for analyses.

2. *Performing analyses on the collected data.* Failure intervals normalized by test activities can be fitted to SRGMs (Section 4) or analyzed by SRGM-DC (Section 6). Input domain and timing information for associated test runs can be analyzed using TBRMs (Section 5). Software tools are needed to support the data processing, analysis, and modeling.

3. *Examining and presenting analysis results.* Analysis results need to be fed back to the software development teams so that appropriate actions can be taken to address the specific problems identified. Appropriate presentation tools need to be selected or constructed to make the interpretation of analysis results easy and to support exploration of alternatives.

Among the various tools that can be adapted and used to support test tracking and analysis activities, there is no single tool that satisfies all the needs for data collection, analysis, and presentation. One option is to construct a comprehensive tool to satisfy all these needs. However, this solution is impractical because of the significant effort required. It is also wasteful because many individual tools have already been used to support various other activities within the development life-cycle. An alternative strategy is to use a collection of loosely integrated tools. Existing tools can be adapted to support some individual needs, and special purpose tools can be constructed for specific applications where no appropriate tool exist. In general, the choice of tools depends on their internal characteristics and external constraints. Important issues include functionality, usability, automation, and flexibility. Since these tools need to work together toward the common goals, various issues regarding the inter-operability and integration also need to be addressed [76].

Central to many of the analyses described in the previous sections is tree-based modeling (TBM), a structured statistical analysis technique that attempts to establish predictive relationships among data attributes [19, 69]. The construction of tree-based models (TBMs) involves recursive partitioning of input data into smaller subsets of increasing homogeneity. The process of recursive partitioning is computationally intensive and needs to be supported by proper software tools. A binary variant of recursive partitioning prescribed in the algorithm in Table 6 (Section 5.1) is supported by a commercial software tool S-PLUS[1]. S-PLUS consists of the S[2] language, a high-level interpretive language [7], and extensive facilities for graphics, data analysis, and statistical modeling [53, 69]. It supports easy visualization of measurement data, regression analysis, and more importantly, TBM. It is the only commercial tool available for TBM. In addition, by using the underlying S language in S-PLUS, special purpose utilities for data processing, reliability modeling and result presentation can be easily developed. This flexibility can also be exploited to integrate the support tool suite. Consequently, S-PLUS was selected as the central tool to support our reliability analysis and improvement activities. Together with other tools for data collection, analysis, and presentation, this forms a comprehensive tool suite [64] discussed below.

Tool Support for Data Collection and Processing

For many large software development organizations, defect data are routinely collected during development using various tools to log defect information and track the resolution and integration of particular defects. In general, such defect tracking tools (or defect logs) record defect data in some underlying databases, from which defect and failure information can be extracted.

These defect logs are used for defect tracking, but not designed to handle timing and input state information. Directly modifying them to record additional test data may interfere with their usage in defect tracking, configuration management and development process control. Consequently, other data gathering tools are generally needed to capture information associated with test execution and failure arrivals, with possible automated support for time-stamping, input state grouping, user prompting, and consistency checking. Our solution to test and reliability data collection is to use test logs, either manual log files, spreadsheets, or semi-automated logging tools.

[1] S-PLUS is a trademark of the Statistical Sciences, Inc.
[2] S is a trademark of the American Telephone and Telegraph Company.

In addition to using test logging tools to collect general input and time domain information, other detailed workload information capturing tools are generally needed if usage dependent data other than simple run counts need to be used. Detailed workload information (e.g., transactions or execution time) during test runs can be captured with the help of such tools or by modifying the program source code. To induce minimal disruption to the testing process, system monitoring tools can be used to gather detailed workload information:

- For execution time measurement, CPU monitoring utilities that come with the operating system can be used to capture the measurements for individual test runs and export them to the test run logging tool.
- For transaction measurement, product or content sensitive monitors are needed. For example, transaction information capturing for product E, a RDBMS products initially studied in [62], is implemented using application program interfaces (APIs) provided with the product, which allow access to monitor information at the database level. This information is logged to disk at predetermined intervals by a background program which is started at the same time as the system test scenario.

The data captured in the data collection tools are the individual test runs and information associated with them, such as run result (success or failure), execution time or transactions extracted from detailed data capturing tools, and test case information for the runs. To use this information for reliability modeling, observed inter-failure intervals (for time-between-failure models, such as the Jelinski-Moranda model [28]) or time periods associated with corresponding number of failures (for period failure count models, such as Goel-Okumoto model [25]) need to be computed (Section 2.3). The time periods are measured in terms of test runs, execution time or transactions. Test runs and associated execution time, transactions or failures can be tallied using utility programs in S language implemented under S-PLUS to compute these normalized time periods and associated failure counts [64].

Individual testing days are commonly used to identify different periods. A testing day is a day where some testing activities occur, as signified by one or more test runs associated with that day. The number of test runs, transactions, and failures can be tallied for each day to use as the input data for reliability modeling. Another advantage of this is that such data index by testing days can be easily used for data grouping, because most of the test activities and management decisions can be associated with calendar days. Often they are *only* associated with calendar days. With this testing day based period indexing, data grouped according to weeks, months, or numerous sub-phases for large projects can be easily derived for reliability modeling to provide modeling results that can be readily used to guide scheduling, product release and other management decisions.

Tool Support for Data Analysis and Modeling

S-PLUS is used to support various key activities, including test tracking, reliability growth modeling, and tree-based reliability modeling:

- *Tracking* of test progress and failure arrivals. This activity includes the visual examination of various entities over time, primarily test runs, workloads, and failures. Utility programs written in S language under S-PLUS were used to tally such entities and visualize their progress.
- *Reliability growth modeling.* Various reliability growth models can be easily implemented in S-PLUS. For example, the Goel-Okumoto model [25] fitted to observed failure data in Figure 9 was produced by nonlinear models in S-PLUS to fit the mean function $m(t) = N(1 - e^{-bt})$, substituting cumulative failures, `cum.fails`, for $m(t)$, and cumulative transactions, `cum.trans`, for t. These data used for model fitting can be calculated from raw data by using data processing utilities implemented in S language under S-PLUS.
- *Tree-based reliability modeling* of reliability using all the time domain and input domain measurement information. S-PLUS is used to construct TBRMs (Section 5) and SRGM-DC (Section 6).

Although some software reliability growth models (SRGMs) can be easily fitted in S-PLUS, most SRGMs contain various options and can use different parameter estimation methods, thus would require significant effort to implement them in S-PLUS. Fortunately, there are various existing tools for reliability growth modeling, including SMERFS (Statistical Modeling and Estimation of Reliability Functions for Software), SRMP (Software Reliability Modeling Program), GOEL (Goel-Okumoto modeling tool), ESTM (Economic Stop Testing Model), AT&T SRE ToolKit, SoRel (Software Reliability Program), CASRE (Computer Aided Software Reliability Estimation), discussed and compared in [52]. Some IBM internal tools for reliability analysis were also compared to some of the above mentioned tools in [59].

Among these reliability modeling tools, SMERFS provides a comprehensive collection of models, runs on multiple platforms, and is widely available to researchers and practitioners [23, 52]. Therefore, it was selected as the center piece for reliability modeling for products studied in this article, which removes the need for implementing many models in S-PLUS. However, SMERFS places strict requirements on the input data format and lacks good graphical presentation of results. Therefore, new facilities were developed to make SMERFS more accessible and usable [58]. Much of the data processing, visualization, and result presentation were automated under S-PLUS. Utility programs written in AWK, S, and C languages were used to extract raw data and convert data into format acceptable to SMERFS.

Tool Support for Result Presentation and Exploration

The measurement data and modeling results can be examined using various visualization techniques supported by S-PLUS. For example, we can visually examine the trends in the observed failure data, plotting cumulative failure arrivals against the normalized time intervals, as well as fitted models, such as in Section 4. SRGMs fitted by S-PLUS or SMERFS can be easily represented in graphical forms in S-PLUS using the automated support program we developed in S. The fitted model objects in S-PLUS contain all the parameters for the models. Alternatively, model parameters can be imported from SMERFS. The modeling results presented in graphical forms such as in Section 4 can be examined visually, and extrapolation of the fitted curve can be used for predictions into the future.

Complete information for the constructed TBMs (TBRMs or SRGM-DC) is stored internally in S-PLUS and can be extracted using built-in and customized utilities. The built-in graphical display of trees draws dendrograms that only show little information for the models: 1) for an interior node, only the split condition is shown; and 2) for a leaf node, only the predicted response value is shown. Such dendrograms were presented to the developers and testers, sometimes with augmented information hand-written on the dendrograms, but a lot of explanations were still needed. Automated support for tree drawing is necessary to produce trees that are easily understandable and visually appealing. The key information that needs to be presented in such trees includes:

- *Node information:* Key information for each node associated with a set of data includes: the size of the data set, the predicted response variable value, and the distribution summary.
- *Branching information:* Each branch represents a binary partition according to some split condition. The essential information needed here is the split condition, which takes two forms: 1) a cutoff value for a selected numerical predictor, or 2) a binary grouping for a selected categorical predictor.

Tree drawings in Section 5 are drawn by new utility programs we implemented in S. Each set of data associated with a tree node is summarized by its predicted success rate (s) and the number of runs (n) in the set. Notice that the response distribution summary is omitted, because the response variable is a binary variable (1 for success and 0 for failure) and the response distribution can be deduced from the set size and predicted response. The split conditions were clearly marked by the edge leading to each partitioned subset. From this kind of tree drawings, we can easily identify nodes with exceptionally low success rates and characterize them by the series of split conditions leading to them from the tree root. Such presentations are easy to interpret and useful to the development teams.

Other information can also be extracted and examined from TBM results in connection with the original input data. For example, we may want to examine the data and distribution of other variables, and examine

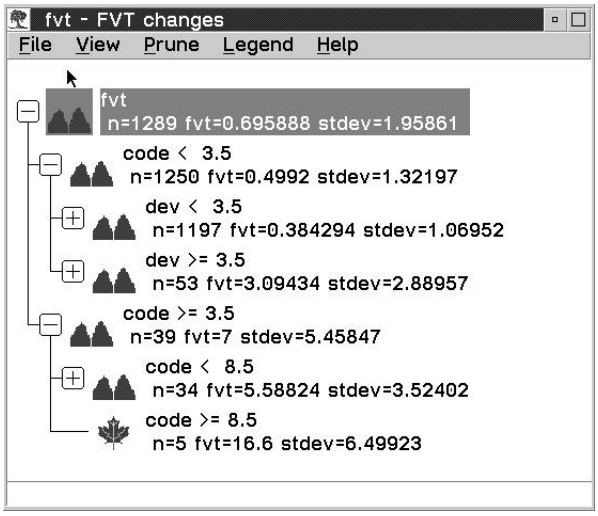


Figure 19: TBM exploration in TreeBrowser

the linkage between the response variable and other predictor variables. However, presenting such information in the static tree drawing like in Figure 11 would be impractical, because too much information will simply ruin the simplicity and direct appeal of such graphs. Some interactive exploration facilities for TBMs need to be considered.

To make the tree exploration easy for developers to use, we recently built an tool called TreeBrowser [67]. The TBMs produced by S-PLUS and the original measurement data are exported to TreeBrowser. TreeBrowser shows the imported models in a hierarchical display. Each tree can be displayed by double clicking on its icon, and its interior nodes can be interactively expanded (by clicking the "+" icon) or pruned (by clicking the "−" icon). Displayed at each tree node is a statistical summary of the subset of data associated with the node and a node type indicator, with a maple leaf icon to indicate a leaf node and an icon with a split in the middle (signaling that it can be further split) to indicate an interior node. This data subset can be displayed in tabular form by double clicking on the tree node. Figure 19 is an example of an interactive display of a TBM in TreeBrowser.

7.3 Integration and Future Development

The tools used for data gathering, analysis, and presentation need to work together to support test tracking and reliability analyses. Figure 20 illustrates the individual tools discussed above and shows their interconnections. Each tool is shown graphically as a rectangle, with the name of the tool identified in boldface letters and the main functions listed under the name. The tools are grouped into three classes: data capturing tools, analysis tools, and presentation tools. The sources for the data capturing tools and the reports produced by the presentation tools are shown in ovals. The interconnections (directed links) show the information flow among the different tools, with the result of one tool used as input to another tool.

As clearly visible from Figure 20, S-PLUS (and its associated S programs) plays a very important role in this tool suite. All the other tools are connected to this block, either as an information consumer (TreeBrowser uses TBM results from S-PLUS, and SMERFS imports formatted data from S-PLUS) or as information providers (SMERFS provides analysis results for presentation, and all other data gathering tools provide raw data for analyses). Consequently, S-PLUS and associated S programs can be used to integrate the tool suite.

In general, tool integration can be achieved primarily through three means:

- *External rules adopted for data contents and formats to ensure inter-operability of tools.* In general,

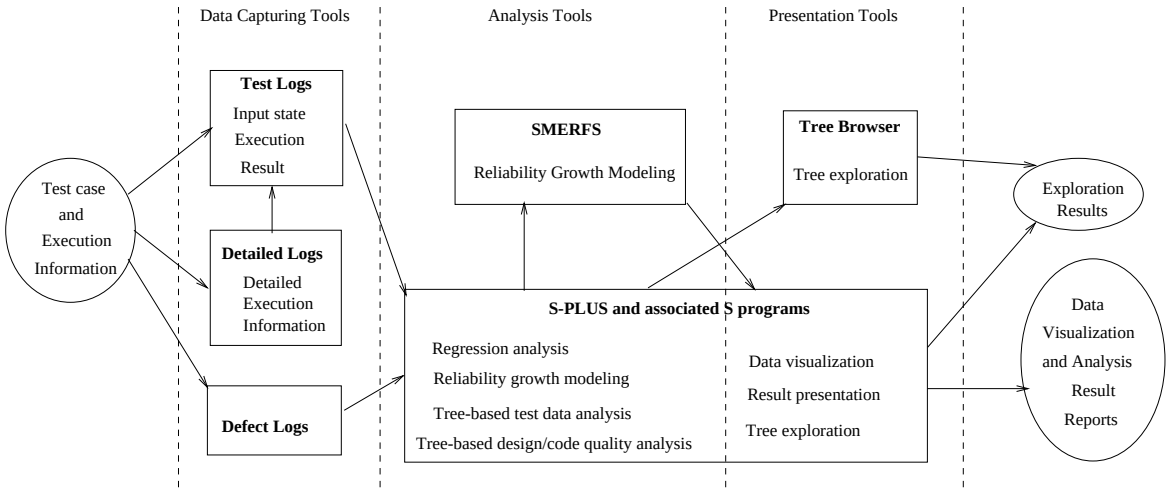


Figure 20: Tools and their connections

before any data collection, analysis, or presentation activity is carried out, all the parties involved have to agree on the data contents and formats. The selection of data contents is primarily determined by the goals of the activities and is prescribed for individual analyses. The format of the data is usually determined by the application environment and the capability of the tools. In general, it is easier to deal with data of different formats in the tools that have greater flexibility. Therefore, data format is generally selected so that it is easy to handle by the less flexible tools.

- *Using common tools for multiple purposes.* For example, S-PLUS, together with the customized S programs, serves multiple purposes and occupies a central place in the tool suite. S-PLUS can be used for some reliability growth modeling instead of SMERFS. In addition, as shown in Figure 20, S-PLUS is connected to all the other tools. It could potentially become the sole integration device, with the user dealing only with S-PLUS while hiding all the other tools in the background. However, this approach would require significant effort and can only be achieved gradually, by incremental additions to utility programs written in S language.

- *Other utility programs that convert data for inter-operability of tools.* For example, various AWK and PERL scripts can be used to extract data for analysis and check for data consistency, which is generally easier to implement than directly processing the raw data in S-PLUS. Some C programs were used to convert failure internal data to the format required by SMERFS for reliability modeling. This collection of small utility programs can be used with integration utilities in S-PLUS to ensure that the tools work together.

To summarize, an integrated implementation and support strategy for various software measurement, analysis, and quality improvement activities have many benefits: collected data, analyses and presentation facilities can be shared among different activities to reduce cost, and integrated analyses can yield results that can not be obtained by individual analyses alone. Careful planning for the implementations that minimize disruption to existing development and testing processes also helps the reception of this integrated strategy in the short term, and the technology transfer in the long term where development organization takes over all the responsibilities for quality and process optimization in addition to product development.

Appropriate tool support is essential to these activities. To accommodate the diverse software measurement environments and data sources, and to support different analyses and usages of the analysis results, a comprehensive suite of tools can be used. The tools can be integrated to work together by observing common data content and format rules, using common tools for multiple purposes, and using utility programs

specifically constructed for tool integration. This approach has been used successfully in supporting software measurement and quality improvement for several large commercial software products developed in the IBM Software Solutions Toronto Laboratory.

There is still much potential waiting to be explored, particularly in the integration of tools so that an unified and easy-to-use environment can emerge. The current users of the tool suite described in this section have to work with several tools and environments. A simplified tool suite where related functionalities are consolidated into fewer tools and most of the standard usage sequences are automated could produce a more homogeneous and easy-to-use tool suite to better support initiatives in software measurement, analysis, and quality improvement.

8 Conclusions and Perspectives

This article presented some recent developments and advances in reliability analysis and improvement for large software systems. Existing analysis techniques were surveyed and discussed. Particular attention was paid to recent developments that not only provide a realistic assessment of product reliability but also help identify problematic areas for focused reliability improvement in a series of studies for large software systems reported in [55, 56, 60, 61, 62, 64]. This article summarized the results from these previous research initiatives, and provided an systematic framework to integrate and share these results. Key observations and findings in this article are summarized below:

- A key ingredient to a successful measurement program is to maximize the utilization of measurement data while minimizing incremental cost and disruption to the existing process. One way to maximize the data utility is to perform an integrated suite of analyses, not only to assess the product quality but also to provide guidance for quality improvement. Visualization techniques and graphical representations can be used to make it easy to interpret and use these analyses results.

- Software reliability growth models (SRGMs) provide an overall quality assessment for products under testing, when proper data and measurement information are captured. Run-based reliability modeling seem to provide the maximal utility with minimal cost, provided that homogeneous runs are used or runs can be broken down to homogeneous sub-runs. Otherwise, time measurement that captures detailed usage information, such as transactions, can be used for reliability analysis with SRGMs.

- Tree-based reliability modeling (TBRM) is a practical technique that can help improve the software development process and product quality. Because of the non-homogeneity of defect distribution within the software products and varied degrees of defect detection effectiveness by different testing scenarios, the identification of specific subsets for remedial actions provides a cost-effective way to improve quality.

- SRGMs based on data clusters (SRGM-DC) is a viable alternative to traditional SRGMs. SRGM-DC generally provide better assessment of current reliability and short term predictions than traditional SRGMs. Long term predictions can be made using the same data clusters as grouped data fitted to traditions SRGMs.

- Integration of various analysis methods provides much more benefit than individual analyses alone. TBRMs and SRGMs complement one another, with the former used *early* in testing to guide actions that lead to improved quality, and the latter used *late* in testing to assess overall quality and to serve as an exit criterion.

- Proper tool support is a key ingredient to a successful implementation of software measurement, analysis and improvement strategies. Sharing and integration of software tools can reduce the cost and provide better support for data collection, analysis and modeling, and result presentation.

Future development of this integrated strategy of reliability measurement and improvement using multiple models and analysis techniques include many of the specific research issues, developments and directions

discussed at the end of previous sections. All these individual developments could help us form a more comprehensive strategy to build effective models for quality, reliability and process improvement and optimization in the testing and development of large software systems.

Acknowledgments

We would like to express our sincere gratitude to: Ian Burrows, Mark Changfoot, George Dadoun, John Henshaw, Ron Holt, Larry Keeling, Peng Lu, Michael Ng, Joe Palma, David Shier, Joel Troster, Joe Wigglesworth, Paul Yee, and members and managers of several testing teams within the IBM Software Solutions Toronto Laboratory. Their support, participation, and feedback are invaluable to our study, particularly to the implementation and deployment of the related software reliability measurement, analysis and improvement activities described in this article. We also thank the editor, Dr. Marvin V. Zelkowitz, for his numerous suggestions which led to a much improved article.

References

[1] ANSI/AIAA. *American National Standard: Recommended Practice for Software Reliability.* Number ANSI/AIAA R-013-1992. American Institute of Aeronautics and Astronautics, Feb. 1992.

[2] ANSI/IEEE. *Standard Glossary of Software Engineering Terminology.* Number STD-729-1991. ANSI/IEEE, 1991.

[3] A. Avritzer and E. J. Weyuker. The automatic generation of load test suites and the assessment of the resulting software. *IEEE Trans. on Software Engineering*, 21(9):705–716, Sept. 1995.

[4] V. R. Basili, G. Caldiera, and G. Cantone. A reference architecture for the component factory. *ACM Trans. on Software Engineering and Methodology*, 1(1):53–80, Jan. 1992.

[5] V. R. Basili and H. D. Rombach. The TAME project: Towards improvement-oriented software environments. *IEEE Trans. on Software Engineering*, 14(6):758–773, June 1988.

[6] V. R. Basili, M. V. Zelkowitz, F. E. McGarry, J. Page, S. Waligora, and R. Pajerski. SEL's software process-improvement program. *IEEE Software*, 12(6):83–87, Nov. 1995.

[7] R. A. Becker, J. M. Chambers, and A. R. Wilks. *The New S Language: A Programming Environment for Data Analysis and Graphics.* Wadsworth & Brooks/Cole, 1988.

[8] B. Beizer. *Software Testing Techniques, 2nd Edition.* International Thomson Computer Press, 1990.

[9] B. Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems.* John Wiley and Son, Inc., 1995.

[10] B. W. Boehm. *Software Engineering Economics.* Prentice Hall, 1981.

[11] B. W. Boehm. Software risk management: principles and practices. *IEEE Software*, 8(1):32–41, Jan. 1991.

[12] L. C. Briand, V. R. Basili, and C. J. Hetmanski. Developing interpretable models with optimal set reduction for identifying high-risk software components. *IEEE Trans. on Software Engineering*, 19(11):1028–1044, Nov. 1993.

[13] W. D. Brooks and R. W. Motley. Analysis of discrete software reliability models. Technical Report RADC-TR-80-84, Rome Air Development Center, Apr. 1980.

[14] J. R. Brown and M. Lipow. Testing for software reliability. In *Proc. Int. Conf. Reliable Software*, pages 518–527, Los Sangeles, CA, Apr. 1975.

[15] D. N. Card and R. L. Glass. *Measuring Software Design Quality.* Prentice Hall, 1990.

[16] M. H. Chen, M. R. Lyu, and W. E. Wong. An empirical study of the correlation between code coverage and reliability estimation. In *Proc. 3rd International Software Metrics Symp.*, pages 133–141, Berlin, Germany, 1996.

[17] M. H. Chen, A. P. Mathur, and V. J. Rego. A case study to investigate sensitivity of reliability estimates to errors in operational profile. In *Proceedings of fifth International Symposium on software reliability engineering*, 1994.

[18] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, and M.-Y. Wong. Orthogonal defect classification — a concept for in-process measurements. *IEEE Trans. on Software Engineering*, 18(11):943–956, Nov. 1992.

[19] L. A. Clark and D. Pregibon. Tree based models. In J. M. Chambers and T. J. Hastie, editors, *Statistical Models in S*, chapter 9, pages 377–419. Wadsworth & Brooks/Cole, 1992.

[20] W. S. Cleveland. *Visualizing Data.* Hobart Press, 1993.

[21] C. J. Date. *An Introduction to Database Systems, Volume 1.* Addison-Wesley, 1990.

[22] W. J. Farr. Software reliability modeling survey. In M. R. Lyu, editor, *Handbook of Software Reliability Engineering*, pages 71–117. McGraw-Hill, New York, 1995.

[23] W. J. Farr and O. D. Smith. Statistical modeling and estimation of reliability functions for software (SMERFS) users guide. Technical Report NSWC TR 84-373, Revision 2, Naval Surface Warfare Center, Mar. 1991.

[24] A. L. Goel. Software reliability models: Assumptions, limitations, and applicability. *IEEE Trans. on Software Engineering*, 11(12):1411–1423, Dec. 1985.

[25] A. L. Goel and K. Okumoto. A time dependent error detection rate model for software reliability and other performance measures. *IEEE Trans. on Reliability*, 28(3):206–211, 1979.

[26] W. Humphrey. *Managing the Software Process.* Addison-Wesley, 1989.

[27] IBM. *Programming Process Architecture, Version 2.1.* IBM, July 1991.

[28] Z. Jelinski and P. L. Moranda. Software reliability research. In W. Freiberger, editor, *Statistical Computer Performance Evaluation*, pages 365–484. Academic Press, 1972.

[29] S. H. Kan, V. R. Basili, and L. N. Shapiro. Software quality: An overview from the perspective of total quality management. *IBM Systems Journal*, 33(1):4–19, 1994.

[30] S. Karlin and H. M. Taylor. *A First Course in Stochastic Processes, Second Edition.* Academic Press, 1975.

[31] B. Kitchenham and S. L. Pfleeger. Software quality: The elusive target. *IEEE Software*, 13(1):12–21, Jan. 1996.

[32] B. Littlewood and J. L. Verrall. A Bayesian reliability growth model for computer software. *Applied Statistics*, 22:332–346, 1973.

[33] P. Lu and J. Tian. Applying software reliability engineering in large-scale software development. In *Proc. 3rd Int. Conf. on Software Quality*, pages 323–330, Lake Tahoe, Nevada, Oct. 1993.

[34] M. R. Lyu, editor. *Handbook of Software Reliability Engineering.* McGraw-Hill, New York, 1995.

[35] R. A. DeMillo, W. M. McCracken, R. J. Martin, and J. F. Passafiume. *Software Testing and Evaluation.* Benjamin/Cummings, 1987.

[36] P. B. Moranda. Prediction of software reliability during debugging. In *Annual Reliability and Maintainability Symp.*, pages 327–332, Washington, DC, Jan. 1975.

[37] J. D. Musa. A theory of software reliability and its application. *IEEE Trans. on Software Engineering*, 1:312–327, 1971.

[38] J. D. Musa. Operational profiles in software reliability engineering. *IEEE Software*, 10(2):14–32, Mar. 1993.

[39] J. D. Musa and W. Ehrlich. Advances in software reliability engineering. In M. V. Zelkowitz, editor, *Advances in Computers, Vol.42*, pages 77–117. Academic Press, 1996.

[40] J. D. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Application.* McGraw-Hill, New York, 1987.

[41] J. D. Musa and K. Okumoto. A logarithmic Poisson execution time model for software reliability measurement. In *Proc. 7th Int. Conf. on Software Engineering*, pages 230–238, Mar. 1984.

[42] E. Nelson. Estimating software reliability from test data. *Microelectronics Reliability*, 17:67–74, 1978.

[43] A. Pasquini, A. Crespo, and P. Matrella. Sensitivity of reliability-growth models to operational profile errors vs testing accuracy. *IEEE Trans. on Reliability*, 45(4):531–540, Dec. 1996.

[44] A. A. Porter and R. W. Selby. Empirically guided software development using metric-based classification trees. *IEEE Software*, 7(2):46–54, Mar. 1990.

[45] B. Ray, I. Bhandari, and R. Chillarege. Reliability growth for typed defects. In *Proc. IEEE Reliability and Maintainability Symp.*, 1991.

[46] R. E. Schafer, J. F. Alter, J. E. Angus, and S. E. Emoto. Validation of software reliability models. Technical Report RADC-TR-79-147, Rome Air Development Center, 1979.

[47] G. J. Schick and R. W. Wolverton. An analysis of competing software reliability models. *IEEE Trans. on Software Engineering*, 4(2):104–120, 1978.

[48] N. F. Schneidewind. Analysis of error processes in computer software. In *Proc. Int. Conf. Reliable Software*, pages 337–346, Los Angeles, California, Apr. 1975.

[49] N. F. Schneidewind. Software reliability model with optimal selection of failure data. *IEEE Trans. on Software Engineering*, 19(11):1095–1104, Nov. 1993.

[50] R. W. Selby and A. A. Porter. Learning from examples: Generation and evaluation of decision trees for software resource analysis. *IEEE Trans. on Software Engineering*, 14(12):1743–1757, Dec. 1988.

[51] M. L. Shooman. Probabilistic models for software reliability prediction. In W. Freidberger, editor, *Statistical Computer Performance Evaluation*, pages 485–502. Academic, New York, 1972.

[52] G. Stark. Software reliability tools. In M. R. Lyu, editor, *Handbook of Software Reliability Engineering*, pages 729–745. McGraw-Hill, New York, 1995.

[53] StatSci. *S-PLUS Reference Manual, Version 3.2.* StatSci, A Division of MathSoft, Inc., Seattle, Washington, Dec. 1993.

[54] R. Thayer, M. Lipow, and E. Nelson. *Software Reliability.* North-Holland, 1978.

[55] J. Tian. Integrating time domain and input domain analyses of software reliability using tree-based models. *IEEE Trans. on Software Engineering*, 21(12):945–958, Dec. 1995.

[56] J. Tian. An integrated approach to test tracking and analysis. *Journal of Systems and Software*, 35(2):127–140, Nov. 1996.

[57] J. Tian and J. Henshaw. Tree-based defect analysis in testing. In *Proc. 4th Int. Conf. on Software Quality*, McLean, Virginia, Oct. 1994.

[58] J. Tian and P. Lu. An integrated environment for software reliability modeling. In *Proc. 17th Int. Computer Software and Applications Conf.*, pages 395–401, Nov. 1993.

[59] J. Tian and P. Lu. Measuring and modeling software reliability: Data, models, tools, and a support environment. Technical Report TR-74.117, IBM PRGS Toronto Laboratory, Mar. 1993.

[60] J. Tian, P. Lu, and J. Palma. Test execution based reliability measurement and modeling for large commercial software. *IEEE Trans. on Software Engineering*, 21(5):405–414, May 1995.

[61] J. Tian and J. Palma. Data partition based reliability modeling. In *Proc. 7th Int. Symp. on Software Reliability Engineering*, pages 354–363, Oct. 1996.

[62] J. Tian and J. Palma. Test workload measurement and reliability analysis for large commercial software systems. *Annals of Software Engineering*, 4:(to appear), Aug. 1997.

[63] J. Tian, A. A. Porter, and M. V. Zelkowitz. An improved classification tree analysis of high cost modules based upon an axiomatic definition of complexity. In *Proc. 3rd Int. Symp. on Software Reliability Engineering*, pages 164–172, Oct. 1992.

[64] J. Tian, J. Troster, and J. Palma. Tool support for software measurement, analysis, and improvement. *Journal of Systems and Software*, 39(2), 1997.

[65] J. Tian and M. V. Zelkowitz. Complexity measure evaluation and selection. *IEEE Trans. on Software Engineering*, 21(8):641–650, Aug. 1995.

[66] J. Troster and J. Tian. Measurement and defect modeling for a legacy software system. *Annals of Software Engineering*, 1:95–118, Sept. 1995.

[67] J. Troster and J. Tian. Exploratory analysis tools for tree-based models in software measurement and analysis. In *Proc. 4th Int'l Symp. on Assessment of Software Tools*, pages 7–17, Toronto, Ontario, Canada, May 1996.

[68] M. Z. Tsoukalas, J. W. Duran, and S. C. Ntafos. On some reliability estimation problems in random and partition testing. *IEEE Trans. on Software Engineering*, 19(7):687–697, July 1993.

[69] W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S-Plus.* Springer-Verlag, 1994.

[70] J. M. Voas and K. W. Miller. Software testability: The new verification. *IEEE Software*, pages 17–28, May 1995.

[71] A. von Mayrhauser. *Software Engineering: Methods and Management.* Academic Press, 1990.

[72] E. J. Weyuker and B. Jeng. Analyzing partition test strategies. *IEEE Trans. on Software Engineering*, 17(7):703–711, July 1991.

[73] J. A. Whittaker and J. H. Poore. Markov analysis of software specifications. *ACM Trans. on Software Engineering and Methodology*, 2(1):93–106, Jan. 1993.

[74] S. Yamada, M. Ohba, and S. Osaki. S-shaped reliability growth modeling for software error detection. *IEEE Trans. on Reliability*, pages 475–478, 1983.

[75] M. V. Zelkowitz. Role of verification in the software specification process. In M. C. Yovits, editor, *Advances in Computers, Vol.36*, pages 43–109. Academic Press, 1993.

[76] M. V. Zelkowitz. Modeling software engineering environment capabilities. *Journal of Systems and Software*, 35(1):3–14, 1996.