Sādhanā, Vol. 11, Parts 1 & 2, October 1987, pp. 23–48. © Printed in India.

A survey of software dependability

V V S SARMA

Department of Computer Science and Automation, Indian Institute of Science, Bangalore 560 012, India

Abstract. This paper presents on overview of the issues in precisely defining, specifying and evaluating the dependability of software, particularly in the context of computer controlled process systems. Dependability is intended to be a generic term embodying various quality factors and is useful for both software and hardware. While the developments in quality assurance and reliability theories have proceeded mostly in independent directions for hardware and software systems, we present here the case for developing a unified framework of dependability–a facet of operational effectiveness of modern technological systems, and develop a hierarchical systems model helpful in clarifying this view.

In the second half of the paper, we survey the models and methods available for measuring and improving software reliability. The nature of software "bugs", the failure history of the software system in the various phases of its lifecycle, the reliability growth in the development phase, estimation of the number of errors remaining in the operational phase, and the complexity of the debugging process have all been considered to varying degrees of detail. We also discuss the notion of software fault-tolerance, methods of achieving the same, and the status of other measures of software dependability such as maintainability, availability and safety.

Keywords. Software dependability; software reliability; software fault-tolerance; computer controlled process systems; software quality assurance.

1. Introduction

A major technological concern for the next decade is the serious and widening gap between the demand for high quality software and its supply. Examples of such systems in the Indian context are the flight control software for the light combat aircraft designed to go into production in the 1990s and the software for the command-control-communication systems of national defence. Process control software for the control and management of nuclear power plants and hazardous chemical processes is also required to be error-free and fault-tolerant.

Computer software refers to computer programs, procedures, rules and possibly associated documentation and data pertaining to the operation of a computer system. System-software pertains to the software designed for a specific computer system or family of systems to facilitate the operation of the computer system and associated programs such as the operating systems, compilers and utilities. Application software is specifically produced for the functional use of a computer, for example, the software for navigation of an aircraft.

Software may conveniently be viewed as an instrument (or a function or a black box) for transforming a discrete set of inputs into a discrete set of outputs. For example, a program contains a set of coded statements which evaluate a mathematical expression or solve a set of equations and store the set of results in a temporary or permanent location, decide which group of statements to execute next or to perform appropriate I/O operations. With a large number of programmers carrying out this task of generating a program, discrepancies arise between what the finished software product does and what the user wants it to do as specified in the original requirement specification. In addition, further problems arise on account of the computing environment in which the software is used. These discrepancies lead to faulty software. Faults in software arise due to a wide variety of causes such as the programmer's misunderstanding of requirements, ignorance of the rules of the computing environment and poor documentation.

Large software systems often involve millions of lines of code, often developed by the cooperative efforts of hundreds of programmers. Enhancing the productivity of software development teams, while assuring the dependability of software, is the challenging goal of software engineering. Problems that come in the way of development of dependable software are: fuzzy and incomplete formulation of system specifications in the initial stages of a software development project, changes in requirement specifications during system development, and imperfect prediction of needed resources and time targets.

A large scale system is often evaluated in terms of its operational effectiveness. The latter is an elusive concept that encompasses technical, economic and behavioural considerations (Bouthonnier & Levis 1984). System dependability is a facet of effectiveness. Dependability is "the quality of service delivered by a computer system, such that reliance can justifiably be placed on this service" (Laprie 1984, 1985). The quality of service denotes its aggregate behaviour characterizing the system's trustworthiness, continuity of operation and its contribution to the plant's trouble-free operation. The behaviour is simply what it does in the course of its normal operation or in the presence of unanticipated undesirable events. In the context of process control, an example of a large scale system is a process controlled by a distributed computer system (DCS). A DCS is defined as a collection of processor-memory pairs connected by a communication subnet and logically integrated in various degrees by a distributed operating system and/or a distributed database. In such a process, the DCS should provide, in real-time, information regarding the plant state variables and structure to the various control agents. The control software must react adequately to the chance occurrences of undesirable events such as physical failures, design faults or environmental conditions. The overall dependability evaluation of the system depends upon the designer's ability to define compatible dependability metrics for the software, hardware and human operator components of a large system.

The paper is organized as follows. Section 2 contains the development of a hierarchical model useful for understanding the effectiveness and the dependability of a complex system in terms of the three basic notions of a system, a mission and a context. Section 3 specializes these definitions to software systems and identifies a set of useful dependability factors. Section 4 presents a detailed study of the software reliability models. Section 5 introduces the notion of software fault-tolerance and describes the means of achieving it. Section 6 briefly reviews the status of other dependability metrics such as availability and maintainability. Section 7 discusses some implications of dependability in the context of process control.

2. A hierarchical model for system evaluation

Figure 1 shows a hierarchical model for defining the effectiveness of a complex system. The operational effectiveness is an elusive concept that encompasses technical, economic and behavioural considerations. Dependability is one facet of a

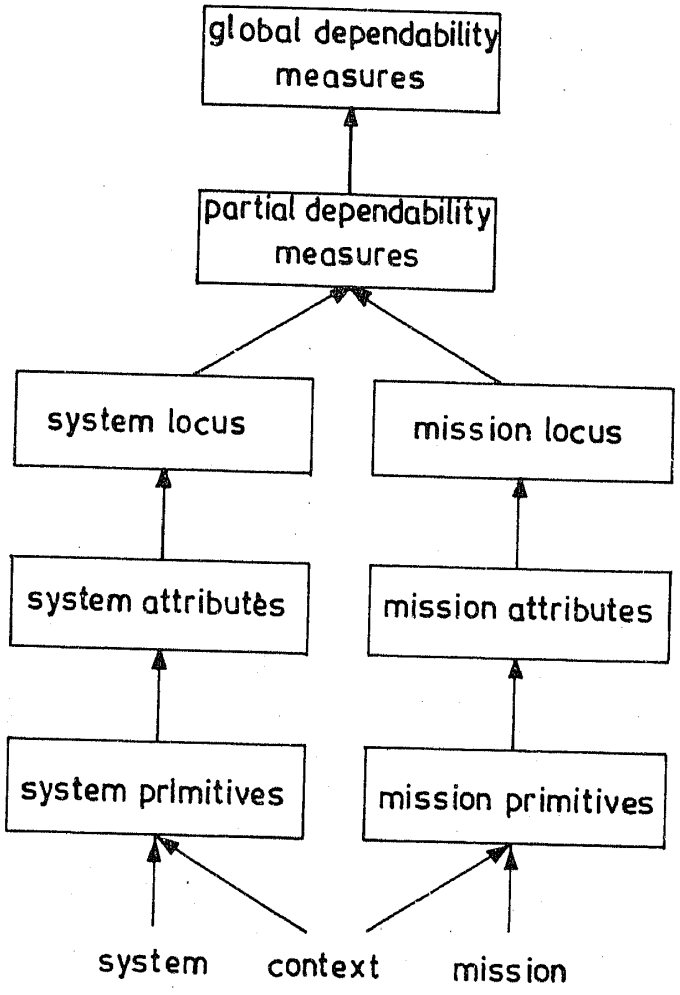

Figure 1. (a) Hierarchical system model for dependability definitions.

System	The whole of the process control system including all of its components (e.g. the plant, the sensors, the actuators, the control computer, including the hardware and the software), the operators, the set of operating procedures and their interactions.
Mission	The set of objectives and tasks that an organization hopes to accomplish with the help of the system over a prescribed time period. The objectives are global accomplishments stated at a hiher level in the hierarchical model. They are achieved by satisfactory completion of lower level functional tasks. If a DCS is a system, transmission of a message between two specified nodes is a particular mission.
Context	The environment in which the mission takes place and the system operates.
Primitives	The parameters that describe the system and the mission. For example, the primitives of a DCS are the numbers of nodes and links, the node reliabilities and the link capacities. The mission primitives are the origin-destination pairs and the message size.
Attributes	Higher level system properties and mission requirements. An example in the DCS context is the maximum delay allowed for communication between a source-destination pair.
Measures of effectiveness	Quantities that result from a comparison of the system and the mission attributes. They reflect the extent to which the mission requirements and the system capabilities match.

Figure 1.(b) Definitions of system effectiveness terms.

system's effectiveness while performance and cost are other important dimensions. Figure 1a defines several of the terms of figure 1 (Bouthonnier & Levis 1984). Example 1 clarifies these terms in the context of a transoceanic flight (a mission) performed by a modern commercial aircraft (a system). Dependability denotes the quality of service delivered by a system as it accomplishes a prescribed mission. By observing this behaviour or the data characterizing it, it is possible to label it as "success" or "failure". There is no need to restrict to binary or dichotomous descriptions. Several levels and combinations of system accomplishments as perceived by interacting systems can be used.

Example 1: (Dependability evaluation of an aircraft flight)
In this example, we consider the effectiveness of an aircraft mission (say, a trans-oceanic flight of a modern transport aircraft) and its relationship to the dependability of its control computers. It is assumed that the computer system is ultra-reliable (with reliability of the order of $1-10^{-9}$ for a 10-hour mission). This level of reliability is achievable in computers such as SIFT (software implemented fault-tolerance) and FTMP (fault-tolerant multi-processor) developed under NASA sponsorship (Siewiorek & Swarz 1982: Viswanadham *et al* 1987). The aircraft mission may be a trans-Atlantic flight from Paris to Houston. Note that the dependability of this particular flight differs for a flight from Paris to New Delhi, which is mostly on land with additional navigational aids and airports for emergency landing. The environment in which the mission takes place determines the context (see figure 1). The system primitives are the aircraft computer components and their reliabilities while the mission primitives are the various

phases of the flight and their durations (Pedar & Sarma 1981). The survivability of the autoland function at the end of a 10-hour flight is a mission attribute while a system attribute is the probability of the loss of a control task due to software errors. Whether we can accomplish the mission on hand with the system may be determined via the system and mission loci which are determined from the corresponding attributes. Using this hierarchical model, it is possible to compute the probabilities of several accomplishment levels achievable by the mission as shown by Pedar & Sarma (1981) (see example 3 below, § 6.3). This also provides a basis for comparing the various fault-tolerant computer architectures for flight control computers.

Example 2 considers the dependability modelling of a complex system consisting of hardware, software and humans.

Example 2: (Overall system reliability assessment)
Let us assume that a good-sized computer system is needed in a critical control application. The first step in system design is to apportion the specified overall system reliability between the hardware, the software and the human operators. An expression for the computer system reliability is given by

$$R = P\ (S.H.O) = P(S)\ P(H|S)\ P(O|H.S). \tag{1}$$

In (1), S, H, and O stand for the events in which the software, the hardware and the operator perform without failure and (.) denotes set intersection. Assuming independence (Shooman 1983),

$$P(H|S) = P(H), \text{ and } P(O|S.H) = P(O), \tag{2}$$

giving

$$R = P(S)\ P(H)\ P(O) = R_S.R_H.R_O. \tag{3}$$

This procedure can be used to set the reliability goals initially. The software, the hardware and the operator can be assumed to be in series. While this example does not throw light on the special characteristics of software reliability, it shows the use of having common dependability measures. A unified framework is thus essential in estimating quantitatively the operational effectiveness and dependability of a large system.

3. Software dependability factors

At present, software dependability is the limiting factor in achieving a high operational effectiveness of complex computer-based systems. Quality assurance has different implications in software and hardware systems. While the emphasis in hardware quality control is on controlling the quality of fabrication of an accepted design, the nature of the design process itself is to be properly understood and controlled for obtaining high quality software. Currently, there is no widely accepted set of factors, definitions or metrics for describing the dependability of software across its lifecycle. Software products and processes may be characterized across many dimensions and levels. At the topmost level, we may specify what are called software dependability factors. This refers to the management-oriented view

of software dependability. Some examples of these factors are: correctness, efficiency, integrity, reliability, maintainability, safety etc. At the middle level, quality attributes from the programmer's viewpoint such as complexity, modularity, security, traceability etc. may also be used to define software dependability. At the lowest level we may consider various metrics and measures which are numbers calculated based on appropriate models of software. These metrics may be related either to the dependability factors at the top level or to the quality attributes at the second level.

In spite of the considerable work in the area of software quality assurance, several questions remain unanswered because of the lack of proper definitions and quantitative information obtained from appropriate data analysis. Some of the questions are (Cavano 1985):

1. How does the software acquisition manager go about establishing meaningful measures for software dependability factors?
2. What tradeoffs need be considered in terms of dependability, cost, schedule and performance?
3. How can future values of factors such as software reliability be predicted and evaluated at key milestones in the development life cycle?
4. What development techniques are required to improve confidence in the project?
5. How much testing should be performed and what testing techniques are required to achieve specified reliability levels?
6. How can the user assess how well dependability goals were met during deployment of the software?

The US Department of Defense (DOD) has sponsored considerable work in the area of software quality and reliability at the RADC (Rome Air Development Centre). What are available today are a large collection of seemingly important software attributes and factors. In the last decade several models have been developed to evaluate the attributes and factors. In this section, we shall briefly review the definitions of some of the factors and survey the state-of-the-art with respect to the extent to which the questions raised above may be answered.

At the first step, we give the preliminary definitions of some software dependability terms as given by the IEEE glossary of Software Engineering Terminology (IEEE 1979).

Software quality

1. The totality of features and characteristics of a software product that bears on its ability to satisfy given needs e.g. to confirm to specifications.
2. The degree to which software possesses a desired combination of attributes.
3. The degree to which a customer or user perceives that software meets his composite expectations.

Quality metric

A quantitative measure of the degree to which the software possesses a given attribute which affects its quality.

Software reliability

1. The ability of a program to perform a required function under stated conditions for a stated period of time.
2. The probability that the software will not cause the failure of a system for a specified time under specified conditions.
This probability is a function of the inputs to and the use of a program as well as a function of the faults existing in the software. The inputs determine whether the faults in a program are encountered in an execution of the program.

Software maintenance

Modification of a software product after delivery to correct latent faults, to improve performance or other attributes or to adapt the product to a changed environment.

Software maintainability

1. A measure of the time required to restore a program to operational state after a failure occurs. Note that in the case of software, service reaccomplishment only requires an execution restart with an input pattern different from the one which led to failure. This measure also depends on whether the software is critical or noncritical (Laprie 1984).
2. The ease with which software can be maintained.
3. Ability to restore the software to a specified state.

Software availability

The probability that the software will be able to perform its designated function when required for use.

Software life cycle

The period of time commencing from the point when a software product is conceived and ending when the product is no longer in use.

The definitions of the terms as presented in the IEEE glossary of terms only indicate the broad sense in which the terms are being used by the software engineering community. The definitions are to be refined considerably, if they are to be of any use in providing quantitative understanding of the field of software dependability. In table 1, we provide a list of dependability factors with their brief descriptions and the phase of the software life cycle in which they can be used.

4. Software reliability

The most widely studied among software dependability factors is software reliability. Definitions 1 and 2 in § 3 characterize the two senses in which the term is used. When used as a metric as per definition 2, the definition should include an appropriate definition of system success or failure, the operational conditions of the software use and the specification of the random variable in question.

Note: At this point, it is helpful to clarify the notions of fault, error and failure. A programmer's mistake is a fault in the system. This leads to an error in the

Table 1. Software dependability factors

Operational phase	
Correctness	Extent to which specifications are met
Reliability	Period in which intended function is met
Efficiency	Amount of computer resources and code needed
Integrity	Extent to which unauthorized access is limited
Usability	Ease of learning and operation
Availability	Fraction of time in which the intended function is met
Safety	Period in which the system does not go to unsafe states
Maintenance phase	
Maintainability	Period in which faults can be located and fixed
Testability	Ease of testing to insure correctness
Flexibility	Ease of modification
Transition phase	
Portability	Transferability from one hardware or software environment to another

written software (e.g. an erroneous instruction or data). The error is latent until activation and becomes effective when an appropriate input pattern activates the erroneous module. This causes deviation in the delivered service (resulting in an unacceptable discrepancy in the output) when a failure is said to occur. Program faults are also called bugs.

The specification of the computing environment must include precise statements regarding the host machine, the operating system and support software, complete ranges of input and output data and the operational procedures. While the conceptual definition is generally accepted, the method of estimating and measuring this quantity is riddled with many questions.

4.1 *Issues in software reliability modelling*

While the definition of software reliability appears similar to its hardware counterpart, several distinguishing features must be carefully considered.

(i) *Phases of software life cycle*: It is convenient to quantify software reliability based on the phase of the software life cycle in which the analysis is conducted (Shooman 1983; Goel 1985). The following phases may be distinguished:

- Development phase
  - Requirements phase
  - Design and programming phase
- Testing phase
  - Module test phase
  - Integration and functional test phase

- Validation phase
- Operational phase
- Maintenance phase
- Retirement/transition phase

Different models and measures of software reliability may be appropriate in different phases of the life cycle.

(ii) *Nature of software development*: The process of software development has considerable effect on the evolution of software reliability notions. Software generation grows through a sequence of less reliable steps. User needs are translated into formal or informal requirements. The requirements are then transformed into formal specifications. These may vary during the development phase resulting in inconsistencies. Further, some of the requirements may involve solutions that are not known or concepts that are not formalizable. In view of these, software dependability depends critically on the reliability of the development process, which cannot be quantified easily.

(iii) *Failure severity classification*: All failures are not identical nor are the bugs causing them. It is convenient to classify software failures as critical, major and minor on the basis of the consequences associated with the failures. An example of a minor failure is a misspelled or badly aligned output and it may just cause annoyance to the user. A major failure may be an irrevocably damaged data base and a critical one is the failure of a control task designed to prevent an accident in a nuclear plant. It may be appropriate to define several software reliability measures such as

$$R_1(t) = P\{\text{no critical failure in interval } [0, t]\}, \tag{4}$$

$$R_2(t) = P\{\text{no critical or major failure in } [0, t]\}, \tag{5a}$$

$$R_3(t) = P\{\text{no critical, major or minor failure in } [0, t]\}. \tag{5b}$$

The reliability measure $R_1(t)$ is easily seen to be a safety measure.

(iv) *Exposure period*: It is often understood that reliability is a perception of the change of a system's quality with time. Hardware reliability is adequately characterized by the random life time (or time-to-failure) of an item. The choice of a suitable random variable is complicated in software reliability. There are many time variables of interest in the software life cycle such as operating time, calendar time during operation, calendar time during development, working time (man-hours) during coding, development, testing and debugging phases and the computer test times throughout the various stages of the program. A possible unit of time in case of an application program is a "run", corresponding to the selection of a point from the input domain (see § 4.2) of the program. The reliability over i runs, $R(i)$, is given by

$$R(i) = P\{\text{no failure over } i \text{ runs}\}. \tag{6}$$

Assuming that inputs are selected according to some probability distribution function, we have

$$R(i) = [R(1)]^i = R^i, \tag{7}$$

where $R = R(1)$. We may define the reliability as follows:

$$R = 1 - \lim_{n \to \infty} (n_f/n), \tag{8}$$

where, n = number of runs and n_f = number of failures in n runs. Several questions arise in using (8) for reliability estimation. In the testing phase, successive runs are to be selected with distinct inputs suitable for exposing certain types of faults as part of a testing strategy. Assumptions are to be made regarding whether modifications are allowed between successive tests after an error is exposed. For some programs (e.g. operating systems), it is difficult to determine what constitutes a run. In such cases, the unit of exposure period is either the calendar time or the CPU time.

$$\begin{aligned} R(t) &= \text{Reliability over } t \text{ seconds} \\ &= P\{\text{no failure in interval } [0, t]\}. \end{aligned} \tag{9}$$

(v) *Structure of software*: A great achievement of the hardware reliability theory is that the system reliability measure incorporates both the stochastic information about component failure behaviour and the deterministic structural information which relates the status of the components and the system in the form of reliability block diagrams, structure functions and fault trees. In contrast, it is more difficult to visualize the components of a program and the structural relationships between the components and the system from a reliability point of view. While the instruction may be viewed as a basic component of a software system, it is more appropriate to think of large programs as composed of separately compilable subprograms called "modules". The complex structure of software does not permit simple relationships between the system reliability, the module reliability and the instruction reliability as in the case of hardware. It is easy to visualize some structural relationships, in case of such constructs as recovery blocks or N-version programming. Additional modelling studies are needed to relate the complexity measures of software with software system reliability. A class of models called micro-models have been proposed to take into account the program path structure in the execution.

(vi) *What leads to a software failure and what are the quantities to be measured?*: These are the fundamental questions to be answered in order to arrive at acceptable software reliability models

A. *Failure modes*: Hardware components normally fail due to the following causes–poor quality of materials and fabrication, overload of components and wear due to old age or wearout. It may be argued that all but the wearout mode apply equally well to hardware and software. The analog of poor quality fabrication is either a typographical error eluding a compiler check or inclusion of the wrong version of a subroutine. Overload occurs because of faster inputting of data at terminals or because of the number of terminals approaching the maximum limit in a time-shared environment. However, most frequently, the software failures are due to man-made design faults unlike hardware systems where such faults are rare. Is the reliability of a program determined by the number of bugs (faults) in it? The early bug counting models of software reliability are based on this view.

B. *Fault seeding*: Estimating the number of remaining faults in a program on the basis of the number of already observed and corrected faults is often not a straightforward proposition. An empirical approach proposed was to seed the program with a number of known faults. The program is tested and the observed number of exposed seeded and indigenous faults are counted. From these an estimate of the fault content of the program prior to seeding is obtained and this is used to estimate the software reliability. The basic assumptions are that seeded faults are distributed uniformly in the program and that both seeded and indigenous faults are equally likely to be detected.

C. *Times between failures*: An alternative to bug count or failure count is to measure the sequence of failure times (or failure intervals). We may assume that the time between successive failures obeys a distribution whose parameters depend upon the number of remaining faults in the program. This provides an alternative framework for model development.

D. *Test inputs*: It is easy to see that a program with two bugs in a little exercised portion of a code is more reliable than a program with only one more frequently encountered bug. In other words, the perceived software reliability depends upon a subset of inputs representative of the operational usage of the program. In § 4.2, we describe a conceptual model of software failures which leads to another class of models (called the input domain based models) for software reliability assessment.

(vii) *Meeting software dependability specifications*: Given that dependability criteria are to be imposed on software systems, there are three main ways which when used together ensure that the required standards are met: (1) fault avoidance–development of design methodologies and environments in which the design faults are eliminated or reduced significantly; (2) fault detection and correction–management of development, testing and validation phases with design reviews, code inspection, program analysis and testing, debugging, verification and data analysis and modelling for dependability prediction; and (3) fault tolerance–employment of defensive programming techniques based on redundancy such as design diversity and multiversion programming.

4.2 *Behaviour characterization of a software system*

The most widely used conceptual model of software is the input-program-output model (Littlewood 1980, see figure 2a). The program P is a mapping from an input space, I, to an output space, O, i.e. $P: I \rightarrow O$. Let O' be the subspace of erroneous outputs, defined with respect to the specification of P, and let I' be the subspace of erroneous inputs such that O' is its image through P. For a (hypothetical) error-free program, the subspace I' will be empty. A failure occurs when the program receives an input from I', which is selected by some (possibly random) mechanism. The situation models the "input uncertainty" leading to system failure (Laprie 1984).

Let us now consider two versions $P(1)$ and $P(2)$ of the same program, written from the same specifications. They have the same input space, I, the same output space, O, and the same subspace of erroneous outputs O'. The two programs differ in the way they partition the input space into a subset I', which will lead to failure and its complement (see figure 2b). The situation shows up the "program uncertainty" component of software unreliability. A sequence of programs $P(1)$,

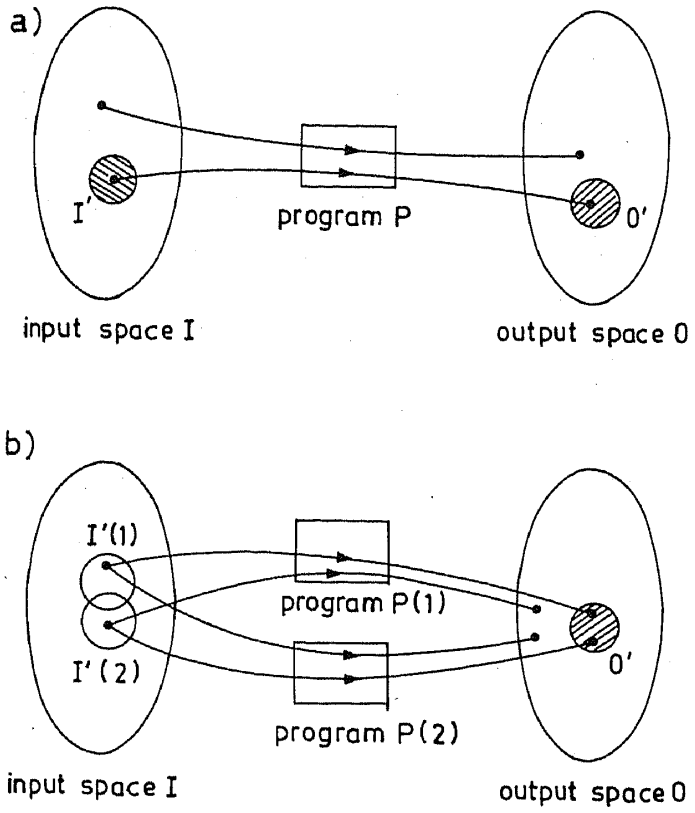

Figure 2. (a) Failure behaviour of a program. (b) Failure behaviour of two versions of the same program.

$P(2), \ldots, P(m), \ldots$, each of them differing from its predecessors by the corrections which have been performed, may now be considered. The result of this debugging process itself is unpredictable and the sequence $I'(1), I'(2), \ldots, I'(m), \ldots$, corresponds to the above sequence of programs. The programmer's intention is to obtain $I'(i) \subset I'(i-1)$ for all $i > 1$, but this cannot be guaranteed. This characterization of the software failure process shows up the clear distinction between (i) the behaviour of the program to failure and (ii) the failure restoration behaviour. This is used in § 4.3d to evaluate the dependability of software in the operational phase. Dependable programs can be defined in this framework. A program may be designed to give specified service in a portion of the input domain and indicate the user when an input from an exceptional domain is encountered.

4.3 *Software reliability models*

The assessment of software reliability is important right from the development phase of the software life cycle. It has to be demonstrated to the user at the time of delivery that the software has the fewest number of faults. Also the cost of detecting and correcting faults via testing increases rapidly with the time to their discovery. A number of models have been proposed in the literature for characterizing (measuring, estimating and predicting) software reliability. The basis of many of these models has often been viewed with considerable skepticism and it has been argued that it is more important to prove that the software meets (or does not meet) its requirements specification. The first difficulty with this latter

approach is that the requirements specification itself may be unreliable, incomplete and may change with time. Secondly, current program verification techniques cannot cope with the size and complexity of software for real-time applications. Exhaustive testing is also ruled out given the large number of possible inputs, limited resources (time and money) allowed for testing, and also because of lack of realistic inputs (e.g. as in a missile defence system). The only feasible approach for assessing software reliability, at present, would appear to be using the existing models.

It is worthwhile to remember that the main aim of modelling is to obtain an abstraction of the behaviour or structure of a real system or a process. There are conflicting requirements such as accuracy, simplicity, ease of validation and ease of data collection in choosing a model and it is also true that any one model is not equally applicable in all conceivable situations. We shall describe several representative models for software reliability in the context of a particular phase of the software life cycle, where they can be most useful (Goel 1984, 1985, Troy & Mosawad 1985).

4.3a *Development phase*: In this phase, the software system is designed as per requirement specification. The program is debugged and tested. Software reliability may be quantified by modelling the process by which the program errors are corrected in the debugging phase. The reliability is related to the number of remaining errors in the program. The early models of Jelinski-Moranda and Shooman belong to this class of error counting models (Shooman 1983). If we assume a perfect debugging process, a previously fixed error does not surface again and a corrective action does not introduce new errors. In such a case, the reliability of a program increases in the development phase, and these models are commonly called reliability growth models. These are used to predict the reliability of the software system on the basis of the error history in the development phase in the form of error count or failure interval data.

Model A. Shooman model of error removal: This models the dynamics of the debugging process. It is assumed that all software errors lead to system failure. No differentiation is made between errors on the basis of their severity or the amount of redesign effort needed to rectify these errors. Data is collected regarding the number of errors removed in the interval $[0, \tau]$, $E_c(\tau)$, where τ is the debugging time in months, and the error removal rate is given by,

$$r(\tau) = \mathrm{d}E_c(\tau)/\mathrm{d}\tau$$

Let I = total number of machine language instructions, then $p(\tau) = r(\tau)/I$ = normalized error removal rate, and $e(\tau) = \int_0^\tau p(x)\mathrm{d}x$ = normalized total number of errors removed. Figure 3 shows the cumulative normalized error curve. We observe that the number of remaining errors is given by

$$E_r(\tau) = E_I - E_c(\tau). \tag{10}$$

Normalizing this by dividing throughout by I, we obtain

$$e_r(\tau) = (E_I/I) - (E_c(\tau)/I) = (E_I/I)\, e_I(\tau). \tag{11}$$

A tempting hypothesis at this stage is to assume that the rate of error detection (and correction) is proportional to the number of errors remaining in the program.

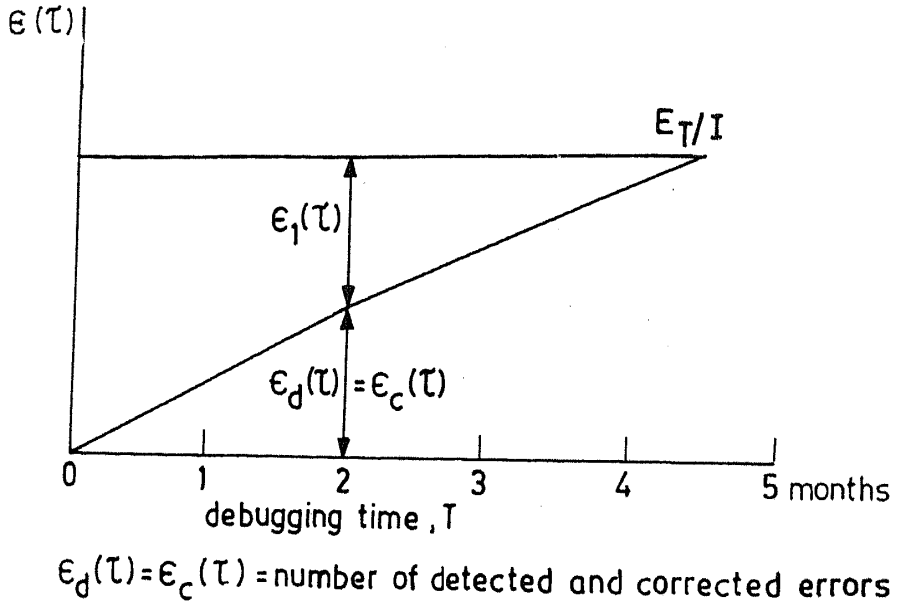

Figure 3. Error removal in the development phase.

$$d\varepsilon_d(\tau)/d\tau = d\varepsilon_c(\tau)d\tau = k[(E_T/I) - \varepsilon_c(\tau)],$$

where k is a constant of proportionality. Hence

$$[d\varepsilon_c(\tau)/d\tau] + K[\varepsilon_c(\tau)] = k[E_T/I] \text{ and } \varepsilon_c(0) = 0. \tag{12}$$

The solution of this equation is

$$\varepsilon_c(\tau) = (E_T/I)\,[1 - e^{-k\tau}]. \tag{13}$$

This is called the exponential error removal model. Several such models can be formulated based on different assumptions of error generation and correction.

We assume that software errors in operation occur because of the occasional traversing of a portion of a program containing a hitherto undetected bug. The hazard rate $h(t)$ is related to the number of remaining bugs in the program, $\varepsilon_1(\tau)$ in (11).

$$h(t) = K\varepsilon_1(\tau) = \gamma, \tag{14}$$

where γ is a constant for a given τ. The corresponding software reliability is

$$R = \exp[-\gamma t]. \tag{15}$$

Model B. The general Poisson model: This is also a bug counting model in which it is assumed that all errors have the same failure rate ϕ. If λ_j is the failure rate of the program after the jth failure, N is the number of bugs originally present, M_j is the number of bugs corrected before the jth failure, and after the $(j-1)$th failure, the failure rate λ_j is given by (Goel 1983).

$$\lambda_j = (N - M_j)\phi. \tag{16}$$

The software reliability is given by the formula

$$R_j(t) = \exp[-\phi(N - M_j)t^{\alpha}]. \tag{17}$$

In (17), α is a constant. In this model, the assumption that all errors have the same failure rate is hard to justify. In fact, earlier errors have a greater failure rate and are detected more easily.

4.3b *Testing phase*: It may be argued that in the final testing and validation phases, the performance of a program as measured by the times between successive failures is more important than its state as measured by the number of residual bugs. Littlewood (1980) proposes a model in which each bug is assumed to cause software failures randomly in a Poisson manner, independent of other bugs in the program.

Model C. Time between failures model: Consider a typical history of software failures as shown in figure 4. T_i is the execution time of the program between the $(i-1)$th and the ith failures and λ_i is the failure rate of the program when $(i-1)$ failures have occurred [i.e. $(i-1)$ bugs removed, leaving $(N-i+1)$ bugs]. If the rate of occurrence of the failures for the jth bug is a random variable ν_j, having the same distribution for all j, the failure rate λ_i of the program is the random variable given by

$$\lambda_i = \nu_1 + \nu_2 + \ldots + \nu_{N-i+1}. \tag{18}$$

$\{\nu_i\}$ is assumed to be a sequence of gamma random variables. The first parameter of the distribution records the way in which bugs in a program are eliminated with certainty when they produce failures. The parameter is of the form $(N-i+1)\alpha$. The second parameter is of the form, $\beta + t_1 + t_2 + \ldots t_{i-1}$, which represents the belief that the bugs which have survived for a long time are, possibly, ones with a small occurrence rate. The distribution of a single bug is gamma with parameters α and β.

The hazard rate, $h(t)$, of the program at the instant marked NOW in figure 4 [i.e. between $(i-1)$th and ith failure] is given by

$$h(t) = [(N-i+1)\alpha]/(\beta + t_1 + t_2 + \ldots + t_{i-1}). \tag{19}$$

Littlewood also suggests estimation of these model parameters using a Bayesian approach.

Model D. Littlewood-Verral model: Successive execution times between failures (see figure 4) are assumed to be exponential random variables, $T_1, T_2, T_3, \ldots, T_i, \ldots$. The density function of T_i is of the form

$$f_{Ti}(t_i|\lambda_i) = \lambda_i \exp(\lambda_i t). \tag{20}$$

The parameters λ_i of the densities are assumed to be independent Gamma random variables, with their density functions defined by

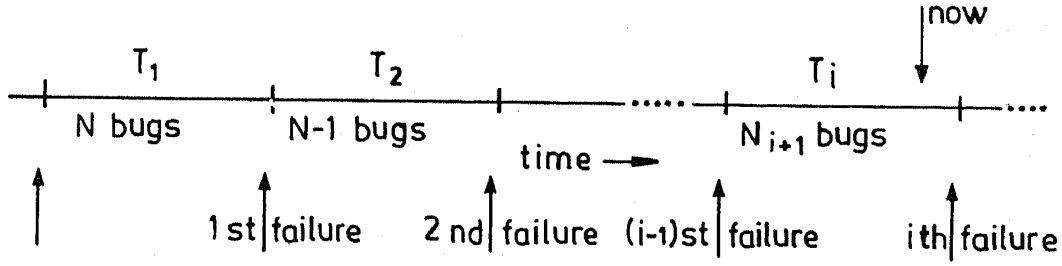

Figure 4. A typical history of failure.

$$f_{T_i}(t_i|\lambda_i) = [\psi_i]^{\alpha}\lambda_i^{\alpha-1}\exp[-\psi(i)\lambda_i]/\Gamma(\alpha). \tag{21}$$

The function $\psi(i)$ is an increasing function of i and it describes the quality of the programmer and the "difficulty" of the programming task. Using (20) and (21), the PDF for T_i which is not directly conditional upon i can be shown to be a Pareto distribution of the form

$$f_{T_i}(t_i|\alpha,\psi(i)) = \alpha[\psi(i)]^{\alpha}/[t_i+\psi(i)]^{\alpha+1}. \tag{22}$$

The reliability growth arising from the debugging process is represented by the growth function, $\psi(i)$. Some method of statistical inference is used to estimate this function $\psi(i)$ and also the parameter α. We may, for example, assume $\psi(i)$ of the form (Littlewood 1980):

$$\psi(i) = \beta_1 + \beta_2 i\,. \tag{23}$$

Thus all the parameters of the models can be estimated from data and the various reliability measures can be derived in a straightforward manner. The Littlewood-Verral model can easily represent reliability decay caused by imperfect debugging as well.

4.3c *Validation phase*: Software used for highly critical real-time control applications must have high reliability. Ideally, we would like to prove that the software meets the specifications. In the validation phase, the software is thoroughly tested by determining the response of a program (success/failure) to a random sample of test cases, specifically for estimating the reliability. The modelling process is used to develop a stopping rule for determining when to discontinue testing and to declare that the software is ready for use at a prescribed reliability level. Any errors discovered in this phase are not corrected. In fact, the software may be rejected if even a single "critical" error is discovered.

Model E. Nelson model: The reliability of software may be estimated by this model in the testing phase (Ramamoorthy & Bastani 1982).

$$R = 1-(n_f/n), \tag{24}$$

where n is the total number of test cases and n_f denotes the number of failures out of these n runs. The estimate converges to the true value of reliability in the limit as n approaches infinity. Such a simple-minded model suffers from several drawbacks, such as (i) a large number of test cases must be used for having a high confidence in reliability estimation, (ii) the approach does not consider any complexity measure of the program such as the number of paths, and (iii) it does not take into account the specific nature of the input domain of the program.

4.3d *Operational phase*: The dependability measures of software systems in the operational phase may be evaluated by using Markov models.

Model F. Behaviour of an atomic system (Laprie-Markov model): Laprie (1984) assumes that an error due to a design fault in an atomic software system produces a failure only if the software system is being executed. He accounts for two types of

processes: (1) the solicitation process, in which the system is alternatively idle and under execution, and (2) the failure process.

Figure 5 shows the Markov model for the atomic software system. In this model, η is the solicitation rate; $1/\eta$ is the mean duration of the idle period. δ is the end-of-solicitation rate; $1/\delta$ is the mean duration of the execution period. λ is the failure rate; $1/\lambda$ is the mean latency of the system. The system reliability $R(t) = P_1(t) + P_2(t)$, where $P_i(t) = P\{\text{system is in state } i \text{ at time } t\}$. $R(t)$ is easily calculated by solving the Markovian state differential equations obtained from figure 5. By assuming that the duration of the execution period is negligible in comparison with error latency, i.e. $\delta \gg \lambda$, and $(\eta + \delta) \gg \lambda$, it may be shown that

$$R(t) = \exp[\eta/(\delta+\eta)] = \exp[-\pi\lambda t],$$

where $$\pi = (1/\delta)/[(1/\delta)+(1/\eta)] \text{ and } \text{MTTF} = 1/\pi\lambda.$$

Model G. Markov model of a complex software system: A complex software system is assumed to be made up of n software components, of failure rates λ_i, $i = 1, 2, \ldots, n$. The transfer of control between components is assumed to be a Markov process, whose parameters are: $1/\delta_i$, mean execution time of component i, $i = 1,2, \ldots, n$ and $q_{ij} = P\{\text{component } j \text{ is executed after component } i | \text{no failure occurred while executing component } i\}$. The Markov model of the system is an $n+1$ state model, where the system is UP in the first n states (component i executed without failure in state i) and the $(n+1)$th state is an absorbing state, the DOWN state. The Markov chain model could be a starting point for reliability modelling of complex software systems (Laprie 1984).

4.3e *Maintenance phase*: In this phase, addition of new features to the software and improvements in the algorithms can be considered. These activities can perturb the reliability of software. Most software reliability models assume that the efficiency of debugging is independent of the system's reliability. Not only is debugging imperfect, but the imperfection increases with time. The longer the system is in operation, the subtler the remaining bugs are and these cannot easily be fixed by a less insightful debugger. Both these things lead to a constant number of bugs in the system after some time. In the maintenance phase, if we combine the continual modification of the program to accommodate new requirements with the subtlety of the remaining bugs and the decreased efficiency of the available debuggers, we can see that the number of bugs increases eventually to a point where the program must be scrapped. With maintenance included in the theory of software reliability, the theory should, therefore, predict that the software does wear out, in the sense that it is no longer economically modifiable.

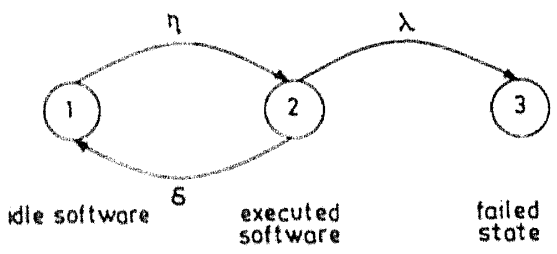

Figure 5. Markov model of the failure behaviour of an atomic software system.

4.3f *Criteria for model comparisons*: The proliferation of software reliability models has caused considerable confusion to software engineers and managers. A decision to choose a particular model in a given context should be based on the following criteria.

(i) *Predictive validity*: The model should be capable of predicting future failure behaviour during either the development phase or the operational phase from the present and the past failure behaviour in the respective phase. The random processes underlying software failures can be described either in terms of the failure intervals (measured in calendar time or execution time) or in terms of counting the errors causing failures. The most general description concerning these processes is given by the probability distribution functions of the several random variables (joint and marginals). Appropriate dependability measures may be obtained from these. At the present time, it appears that error counting approaches are more practical for use (Troy & Moawad 1985).

(ii) *Capability or utility*: The capability or the utility of a model should be judged on the basis of its ability to give sufficiently accurate estimates of quantities such as the present reliability, the MTTF, the number of residual faults, the expected date of reaching a specified reliability goal, the expected cost for reaching a dependability goal and the estimates of human and computer resources needed to achieve this goal.

(iii) *Quality and modelling assumptions*: Modelling involves many simplifying assumptions as it attempts an abstraction of a real process. These assumptions need to be validated based on the data available. While the axiomatic theories adopt the Leibnizian philosophy "truth is in the model", statisticians take a Lockean approach "truth is in the data". We believe that the Kantian statement, "truth is not entirely in a model or in data but emerges from the interaction of model and data", should be the natural choice of the software engineer in selecting a software reliability model. The process of enquiry gets into a cycle of hypothesize-measure-analyse-predict steps (Churchman 1971). Troy & Moawad (1985) recommend the validation of modelling assumptions at conceptual, structural and operational levels and compare the models on the basis of the components of inputs, outputs, estimators of parameters, hypotheses involved and mathematical formulation.

(iv) *Applicability*: A model should be judged on the basis of the degree of applicability across different software products (size, structure, function), different operational environments, different life cycle phases and different types of reliability behaviour (reliability growth and decay).

(v) *Simplicity*: The model should be conceptually simple and should permit inexpensive data collection.

(vi) *Program complexity and structure*: It is reasonable to assume that the software cannot be treated as "atomic", meaning that it is an indivisible unit. The structure of the software system, composed of separate units or modules, is to be taken into account in a more meaningful way than is being done presently. The system structure changes during the development phase, and dependability modelling should take into account the evolution of the system structure.

The study of Troy & Moawad (1985) represents a step in the direction of a systematic comparison of the well-known Musa (1984) and Littlewood-Verrall models on the basis of the criteria listed above using RADC/DAC database. Goel (1985) recommends a step-by-step procedure for developing and choosing

Table 2. Steps in software reliability assessment

Step	Activity
1	Study software failure data
2	Choose a reliability model based on the life cycle phase
3	Estimate model parameters using a method such as maximum likelihood estimation
4	Obtain the fitted model
5	Perform goodness-of-fit test
6	Go to step 2 if the test rejects the model and repeat through step 6 with another model or after collecting additional data
7	Decision making regarding the release of the software product or continuation of the testing process

appropriate models for software reliability assessment. Table 2 summarizes these steps.

5. Software fault-tolerance

The tolerance of design faults, especially in software, is a more recent addition to the objectives of fault-tolerant system design. The two currently identified strategies of achieving fault-tolerance in software systems are recovery blocks and design diversity.

5.1 *Recovery blocks*

A block of a program is any segment of a large program (e.g. a module, a subroutine, a procedure or a paragraph) which performs some conceptual operation. Such blocks also provide natural units for error-checking and recovery. By adding extra information for this purpose they become recovery blocks. The basic structure of a recovery block is shown in figure 6. A recovery block consists of an ordinary block in the programming language (the primary alternate), plus an acceptance test and a sequence of alternate blocks. The acceptance test is just a logical expression that is to be evaluated upon completion of any alternate to determine whether it has performed acceptably. If an alternate fails to complete (e.g. because of an error or exceeding of a time limit) or fails the acceptance test, the next alternate (if any) is entered. However, before a further alternate is tried, the state is restored to what it was just prior to entering the primary alternate.

```
A: ensure T
     by P : begin
            -- program text --
            end
else by Q : begin
            -- program text --
            end
  else error
```

Figure 6. Recovery block structure (T: acceptance test, P: primary alternate and Q: secondary alternate)

When no further alternates remain after a failure, the recovery block itself is deemed to have failed (Randell 1975).

Laprie (1984) models the design process of a recovery block, and evaluates its reliability in a Markovian framework. He notes the important role played by the acceptance test in the failure process of a recovery block. It is not hard to give a structural reliability interpretation for this. A recovery block with three alternates is analogous to a system with triple modular redundancy (TMR). The acceptance test is a critical component of the structure similar to that of a voter in a TMR system.

5.2 *Multiple computations*

The use of redundant copies of hardware, data and programs has proven to be quite effective in the detection of physical faults and subsequent recovery. This approach is not appropriate for tolerance of design faults that are due to human mistakes or defective design tools. The faults are reproduced when redundant copies are made.

A. *Design diversity*: Design diversity is the approach in which the hardware and software elements that are due to be used for multiple computations are not copies, but are independently designed to meet the given requirements. Different designers and tools are employed in each effort and commonalities are systematically avoided. The obvious advantage of design diversity is that reliable computing does not require the complete absence of design faults, but that designers should not produce similar errors in the majority of designs.

A very effective approach to the implementation of fault-tolerance has been the execution of multiple (N-fold with $N \geq 2$) computations that have the same objective: a set of results derived from a given set of initial conditions and inputs. Two fundamental requirements apply to such multiple computations: 1) The consistency of initial conditions and inputs for all N instances of computation must be assured, and 2) A reliable decision algorithm that determines a single decision result from the multiple results must be provided.

The decision algorithm may utilize a subset of all N results for a decision; e.g., the first result that passes an acceptance test may be chosen. The decision may also be that a decision rule cannot be determined, in which case a higher level recovery procedure may be invoked. The decision algorithm itself is often implemented N times – once for each instance of computation that uses the decision result. Only one computation is then affected by the failure of one decision element (e.g., a majority vote). Multiple computations may be implemented by N-fold replications in three domains: time (repetition), space (hardware) and program (software). Notation: The reference or simplex case is that of one execution (simplex time $1T$), of one program (simplex software $1S$) on one hardware channel (simplex hardware $1H$), described by the notation: $1T/1S/1H$. Concurrent execution of N copies of program on N hardware channels is $1T/NS/NH$. Examples of such systems are the space-shuttle computer ($N = 4$), SIFT ($N \geq 3$), FTMP ($N = 3$) and No. 1 ESS (Electronic Switching System – AT & T Bell Laboratories) ($N = 2$) (Siewiorek & Swarz 1982). The N-fold time cases employ the sequential execution of more than one computation. Some transient faults are detected by the repeated execution of the same copy of the program on the same machine ($2T/1S/1H$).

Faults that affect only one in a set of N computations are called "simplex" faults.

A simplex fault does not affect other computations although it may be a single or a multiple fault within one channel. Simplex faults are effectively tolerated by the multiple computation approach as long as input consistency and an adequate decision algorithm are provided. The M-plex faults ($2 \leq M \leq N$) that affect M out of the N computations form a set for the purpose of fault-tolerance. Related M-plex faults are those for which a common cause that affects M computations exists or is hypothesized.

B. *N-version programming*: An effective method to avoid identical errors that are due to design faults in N-fold implementations is to use independently designed software and/or hardware elements instead of identical copies. This approach directly applies to parallel systems *1T/NDS/NH*, which can be converted to *1T/NDS/NH* for N-fold diverse software or *1T/NS/NDH* for N-fold diverse hardware of *1T/NDS/NDH* for diversity in both hardware and software. The sequential systems (N-fold time) have been implemented as recovery blocks with N sequentially applicable alternate programs (*NT/NDS/1H*) that use the same hardware. An acceptance test is performed for fault detection and the decision algorithm selects the first set of results that pass the test.

N-version programming is the independent generation of $N \geq 2$ software modules called "versions", from the initial specification. "Independent generation" here means programming efforts carried out by individuals or groups which do not interact with respect to the programming process. Wherever possible, different algorithms and programming languages or translators are used in each effort. The goal of the initial specification is to state the functional requirements completely, while leaving the widest choice of implementations to the N programming efforts. An initial specification defines: (1) the function to be implemented by an N-version software unit, (2) data formats for the special mechanisms: decision vectors (*d*-vectors), decision status indicators (*ds*-indicators), and synchronization mechanisms, (3) the cross-check points (*cc*-points) for *d*-vector generation, (4) the decision algorithm, and (5) the responses to the possible outcomes of decisions. The term decision is used here as a general term, while comparison refers to the $N = 2$ case and the term voting to a majority decision with $N > 2$. The decision algorithm explicitly states the allowable range of discrepancy in the numerical results.

It is the fundamental conjecture of the N-version approach that the independence of programming efforts will greatly reduce the probability that software design faults will cause similar errors in two or more versions. Together with a reasonable choice of *d*-vectors and *cc*-points, the independence of design faults is expected to turn N-version programming into an effective method to achieve design fault-tolerance. The effectiveness of the entire approach depends on the validity of this conjecture. Experimental investigations are necessary to demonstrate this validity. The following questions remain to be answered: (1) Which requirements (e.g., need for formal specifications, choice of suitable type of problems, nature of algorithms, timing constraints, decision algorithms etc.) have to be met for the N-version programming to be feasible at all, regardless of its cost? (2) What methods should be used to compare the cost and the effectiveness of this approach to the two alternatives: single version programming and the recovery block approach?

Avizienis (1985) discusses these issues and the experiments at UCLA. McHugh

(1984) describes a large scale experiment conducted by several universities (North Carolina State University, University of California at Los Angeles, University of Illinois, and University of Virginia) to determine the effect of fault-tolerant software techniques under carefully controlled conditions. Graduate students from these four places produced multiple versions of the same function and the resulting code was combined in a variety of fault-tolerant configurations such as recovery blocks and *N*-version schemes. The idea is to avoid correlated errors in the codes produced. A surprising outcome was that programmers at two universities generated identical faults. Multiversion software has been proposed for use in the safety control system of nuclear reactors. The outputs of these multiple versions are operationally subjected to a majority voter and the system gives incorrect outputs whenever a majority of its component versions fail. An unexpected outcome from recent experimental studies is that totally uncorrelated design faults in the software appeared as coincident failures in the application environment with two or more versions failing when operating on the same input. It is therefore necessary to study conditions under which employment of multiversion software is a better strategy than use of a single version (Eckhardt & Lee 1985).

C. Consensus protocols: Voting is used by a wide variety of algorithms in distributed systems to achieve mutual exclusion. For example, consider a restricted operation like updating a database. Each computer or node participating in the algorithm is given a number of votes out of a total of T votes. Only a group of nodes with a majority of votes performs the updating. Similarly in commit protocols, voting ensures that a transaction is not committed by one group and aborted by another. It is possible to optimize the reliability of critical operations provided by voting mechanisms in DCS by using consensus protocols. These involve optimal vote assignment after efficiently partitioning the network into groups of nodes (Garcia-Molina & Barbara 1984).

6. Other dependability factors

6.1 *Software maintainability*

A question often asked in software engineering is "How is a software system modularized so that the resulting modules are both testable and maintainable?" The issues of testability and maintainability are important dimensions of dependability if we recognize the fact that half of development time is spent in testing while maintainance has the potential of absorbing a considerable portion of the budget. Thus testability is important in the design and testing phases while maintainability is important in the operational and maintainance phases of the software life cycle.

A hardware system is usually visualized as performing an alternating movement between two states, UP and DOWN, separated by the events of failure and representation. While reliability is the measure of the random time duration during which the system is continuously in the UP state, maintainability refers to the time spent in DOWN state. MTTR (mean time to repair) characterizes maintainability for software systems. Laprie (1984) recommends a similar model for software (valid, at least, for critical software). Characterizing the maintainability of a general software

is complexity. One approach suggested in the literature is based on using the ed to metrics of a program. The complexity is a measure of the effort program understand the program and is usually based on the control or data flow clear am (McCabe 1976). The self-descriptiveness of a program is a measure res the program is, i.e., how easy it is to read, understand and use. Other t such as extensibility (a measure of the extent to which the program can the extension of critical functions), stability (resistance to amplification of s), accessibility (ease of access of a program module) and testability (effort d for testing) have also been used to assess a program's maintainability. of these measures can be defined using the representation of the program as ted graph (Mohanty 1979). An alternative approach which shows promise is e at a measure of program difficulty as a function (e.g. a simple sum) of the ties of its constitutent elements. Berns uses this approach to arrive at the ty measure of a FORTRAN program by assigning weights to its elements (e.g. ent types, operators, symbols etc.) (Berns 1984).

ftware availability

the definition of availability as given in § 3 appears satisfactory, its ion poses some problems for software. The failure-restoration model ed by Laprie (1984) appears justifiable only in the case of critical software. of noncritical software, correction of errors can be performed on a different f the software. The classical availability expression

$$A = \text{MTTF}/(\text{MTTF}+\text{MTTR})$$

efore applicable in the case of either critical software or in the early oment phase.

ftware safety

creased use of computers in life critical applications has brought importance atively unexplored facet of software development–software safety (Leveson We may define safety as follows.

ion: Software safety is the probability that a given software system operates ne time period without an accident resulting in injury or death to humans ing with the system or unintentional damage to the equipment or data caused by a software error on the machine or the distributed environment ch it is designed.

system safety is a global dependability measure (figure 1). To determine ormally, appropriate mission accomplishments are to be defined. We may at every failure is not safety-related. Only a subset of failure states are of as seen from the following example.

e 3 (Safety of an aircraft flight):
evaluating the performability of an aircraft, it is usual to consider the ng accomplishment set (Pedar & Sarma 1981):

$$A = \{a_i | i = 1,2,3,4,5\}.$$

Each a_i denotes a distinguishable level of accomplishment, given by

a_1 = a fuel-efficient and safe mission from the source to a destination airport,

a_2 = a safe mission from source to destination but the fuel management task has failed,

a_3 = a fuel-efficient flight but the flight is diverted to a different airport because the autolanding failed,

a_4 = a safe mission with diversion and with failures of fuel management function,

a_5 = a fatal crash.

The performability is a combined measure incorporating performance and reliability and is used for degradable computer systems. In this example, only the accomplishment level denoting fatal crash is related to the safety of the mission.

In the absence of adequate safety specifications, even with the system operating satisfactorily with respect to availability and performance specifications, the system might be operating in accident-prone conditions. Safety is of prime concern both during the normal operation of the system and in the presence of undesirable events such as module failures and environmental causes.

7. Dependability issues in real-time process control

A control system, or process, can be divided into the controller and the controlled plant. A control design job can be broadly defined as specifying and implementing the functions that drive the inputs so that a plant performs a specified process (or completes a prescribed mission). Design may be broken into two successive tasks, control engineering and software engineering.

The dynamics of the plant are described by the state equations, relating the inputs, the outputs and the internal state variables of the plant. The controller often includes a model of the plant (either an observer or a Kalman filter necessary for implementing optimal and adaptive control functions). The determination of the control law is in the domain of control engineering. Software engineering implements the specified control law as an executable computer program, accommodating such criteria as dependability and flexibility.

The control theory component refers to the feedback control action of moderate complexity and high reliability. The needed response times are from seconds to minutes in the process control context. In contrast, the man/machine system is of the open loop control variety and deals with process variable display systems and alarm systems. The possible states and transitions are shown in figure 7. The required response times are tens of minutes. The fault-tolerant trip systems are ultra-reliable, closed-loop control systems which initiate drastic actions such as the fail-safe plant shut down. The ability of the process to either survive in the face of failures, errors, and design flaws or to be safely shut down is to be evaluated. Appropriate measures for dependability specification and evaluation of process

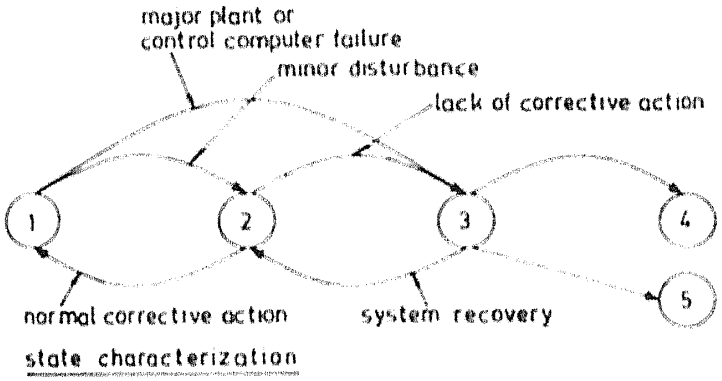

Figure 7. State transitions in process control systems

control systems and associated control software can be defined only by complete mathematical modelling of the process, including the plant, and the controller, including the implementation of the control law (Viswanadham *et al* 1987).

All the control functions are to be taken into account in the evaluation of software dependability measures in the process control context. It is assumed in the subsequent analysis that all these are implemented in software. The basic control software tasks involve the continuous time process control algorithms. An additional set of software tasks include the open-loop functions of data instrumentation, data logging and alarm systems for the man-machine subsystem. The critical set of software tasks involve software-based safety (trip) procedures involving safe shut down of the process. Software safety study essentially involves reliability evaluation, verification, and validation of this particular software task.

The main issue in the design of software is to establish the safety boundary between the control in the normal or minor upset operating region and the automated protection system. This decision regarding the operating region is based on multi-sensor data evolving in time. The decision procedures invariably involve some type of sequential probability tests (SPRT). As in any hypothesis testing situation, there are chances of two types of errors (the miss and the false alarm) with different costs of consequences. The miss may mean an increased risk of a potential disaster while a false alarm might mean economic penalty due to unwarranted plant shut down. In some cases, it may be worthwhile to delay the decision by a small amount of time so that further data may be accumulated. System level safety measures include the reliability of this decision making capability.

References

Avizienis A 1985 *IEEE Trans. Software Eng.* SE-11 1491–1501
Berns G M 1984 *Commun. ACM* 27 11–23
Bouthonnier V, Levis A H 1984 *IEEE Trans. Syst. Man Cybern.* SMC-14 48–53
Cavano J P 1985 *IEEE Trans. Software Eng.* SE-11 1449–1455
Churchman C W 1971 *The design of inquiring systems* (New York: Basic Books)
Eckhardt D E, Lee L D 1985 *IEEE Trans. Software Eng.* SE-11 1511–1517

Garcia-Molina H, Barbara D 1984 *Proc. 4th Int. Conf. on Distributed Computing Systems* (Silver Spring, MD: IEEE Comput. Soc. Press) pp. 340–346
Goel A 1983 A Guidebook for Software Reliability Assessment, Tech. Rept. 83–11 (New York, Syracuse Univ. Dept. Ind. Eng. & OR)
Goel A 1985 *IEEE Trans. Software Eng.* SE-11: 1411–1423
IEEE 1979 *Software engineering terminology* (New York: IEEE Press)
Laprie J C 1984 *IEEE Trans. Software Eng.* SE-10: 701–714
Laprie J C 1985 *Digest Fault-tolerant Comput. Symp.* (Silver Spring, MD: IEEE Comput. Soc. Press) 15: pp. 2–11
Liveson N G 1984 *Computer* 17: 48–55
Littlewood B 1980 *IEEE Trans. Software Eng.* SE-6: 480–500
McCabe T J 1976 *IEEE Trans. Software Eng.* SE-2: 308–320
McHugh J 1984 *Proc. Seventh Minnowbrook Workshop on Software Performance Evaluation*, Syracuse University, New York, pp. 73–74
Mohanty S N 1979 *ACM Comput. Surv.* 11: 251–275
Musa J 1984 in *Handbook of software engineering* (eds) C R Vick, C V Ramamoorthy (New York: Van Nostrand) chap. 18, pp. 392–412
Pedar A, Sarma V V S 1981 *IEEE Trans. Reliab.* R-30: 429–437
Ramamoorthy C V, Bastani F B 1982 *IEEE Trans. Software Eng.* SE-8: 354–371
Randell B 1975 *IEEE Trans. Software Eng.* SE-1: 220–232
Shooman M L 1983 *Software engineering* (New York: McGraw-Hill)
Siewiorek D P, Swarz S 1982 *The theory and practice of reliable system design* (Bedford, Mass: Digital Press)
Troy R, Moawad R 1985 *IEEE Trans. Software Eng.* SE-11: 839–849
Viswanadham N, Sarma V V S, Singh M G 1987 *Reliability of computer and control systems* (Amsterdam: North Holland)