Chapter

8

Measurement-Based Analysis of Software Reliability

Ravishankar K. Iyer
University of Illinois

Inhwan Lee
Tandem Computers Inc.

8.1 Introduction

Software reliability and quality must be built in, starting in the early design phase, and maintained throughout the software life cycle. Essential to this process is a sound understanding of software reliability in production environments. There is no better way to acquire this understanding than through the direct measurement and analysis of real systems. Direct measurement means monitoring and recording naturally occurring errors and failures in a running system under user workloads. Analysis of such measurements can provide valuable information on actual error/failure behavior, identify system bottlenecks, quantify reliability measures, and verify assumptions made in analytical models.

Typically, a software engineer must decide what data to gather and analyze, sometimes without the benefit of guidance, experience, or easily available intuition. How to obtain general models from experiments or measurements made in a particular environment is by no means clear. This chapter discusses the current issues in this area. The discussion centers around techniques, our experiences, and major developments. The chapter discusses measurement techniques, analysis of data, model identification, analysis of models, and the effects of workload on software reliability. For each field, the key issues are discussed and then detailed techniques and representative work are presented. Analytical modeling techniques and statistical techniques relevant to the discussions are reviewed in App. B.

8.2 Framework

This section discusses the framework of measurement-based analysis and reviews past work in the area of software reliability in the operational phase.

8.2.1 Overview

Once general directions are set, a measurement-based study consists of two phases: measurement and analysis (Fig. 8.1). In the measurement phase, you develop instrumentation techniques and make measurements. You can conduct a measurement-based study of operational software using two types of data: human-generated software error reports and machine-generated event logs. The former provide detailed information about the underlying software faults (or defects) and the associated failure symptoms, while the latter provide accurate information on the timing of software failures and recovery. Measurement techniques are discussed in Sec. 8.3.

Given field error data collected from a real system, the analysis consists of five steps, as shown in Fig. 8.1*b:* (1) preprocessing of data, (2) analysis of data, (3) model structure identification and parameter estimation, (4) model solution, if necessary, and (5) analysis of models.

In step 1, you extract necessary information from the field data. The processing in this step requires detailed understanding of the target software. It can also require detailed knowledge of the operating system and system operation. The actual processing depends on the types of data. The information in human-generated reports is

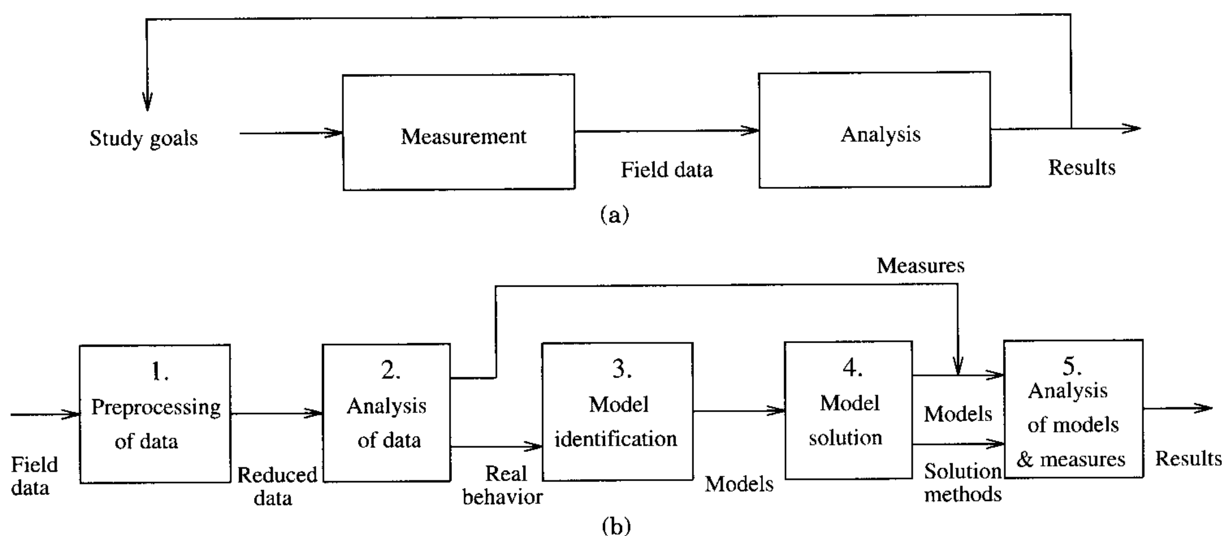


Figure 8.1 Measurement-based analysis: (*a*) overall framework; (*b*) analysis phase.

usually not completely formatted. Therefore, this step involves understanding the situations described in the reports and organizing the relevant information into a problem database. In contrast, the information in automatically generated event logs is already formatted. Data processing of event logs consists of extracting error events and coalescing related error events. Section 8.4.1 discusses the preprocessing of data.

In step 2, you interpret the data. Typically, you begin this step with a list of measures to evaluate. However, you can identify new issues that have a major impact on software reliability during this step. Results from step 2 are reliability characteristics of operational software in actual environments and issues that must be addressed to improve software reliability. Sections 8.4 and 8.5, which cover this step, discuss fault and error classification, error propagation, error and failure distribution, software failure dependency, hardware-related software errors, evaluation of software fault tolerance, error recurrence, and diagnosis of recurrences.

In step 3, you identify appropriate models (such as Markov models) based on the findings in step 2. You identify model structures and realistic ranges of parameters. Identified models are abstractions of the software reliability behavior in real environments. Proposed software reliability models include: performability models [Hsue88, Lee93a], an error and recovery model [Hsue87], a software reliability model that captures the effects of faults on the overall system [Lee94b], and workload-dependent software reliability models [Cast81, Iyer82a]. Statistical analysis packages such as SAS [SAS85] or measurement-based reliability analysis tools such as MEASURE+ [Tang93b] are useful at this stage. Step 3 is covered in Secs. 8.6 and 8.7.

Step 4 involves either developing or using known techniques to solve the model. Model solution allows you to obtain measures, such as reliability, availability, and performability. The results obtained from the model must be validated against real data. You can use reliability and performance modeling and evaluation tools such as SHARPE [Sahn87] in this step. In step 5, you answer what-if questions, using the identified models. You vary factors in the models and evaluate the resulting effects on software reliability. You determine reliability bottlenecks and predict the impact of design changes on software reliability. Section 8.6, which covers this step, discusses software reliability modeling in the operational phase and the modeling of the impact of software failures on performance, detailed error and recovery processes, and software error bursts. You use knowledge and experience gained through analysis to plan additional studies and to develop the measurement techniques as shown in Fig. 8.1*a*.

8.2.2 Operational versus development phase evaluation

Figure 8.2 shows a simplified software life cycle. To construct new software or to add a new feature, you begin with requirements and then design, implement, and verify the software. After verification, the software is released to the field. Problems found in the field are diagnosed, fixes are made, and interim versions of the software are released to the field. As a result, many versions of the same software exist in the field at the same time. The process in Fig. 8.2 is repeated until the software becomes obsolete.

You can perform an experimental evaluation of software reliability at different phases of the software's life. In the development phase, data are generated as a result of code inspection and software testing. Many studies have addressed the evaluation of data collected during the development stage. However, the reliability of operational software can be quite different from that of the software in its development stage. In the operational phase, the software update rate is relatively low. Due to the differences among the fault severities (i.e., their impact on system functionality) and due to software fault tolerance features in a system, not every software fault has the same impact on software reliability. Workloads, interaction between software and hardware platforms, and operational environment are also factors that affect reliability. Thus, you cannot accurately estimate software reliability in the operational phase using only the data collected during the development phase. Understanding the reliability of software requires direct measurement during the operational phase.

8.2.3 Past work

Measurement-based reliability analysis of operational software has evolved significantly over the past 15 years. Many studies have been published. Table 8.1 lists some of the studies that are closely related to

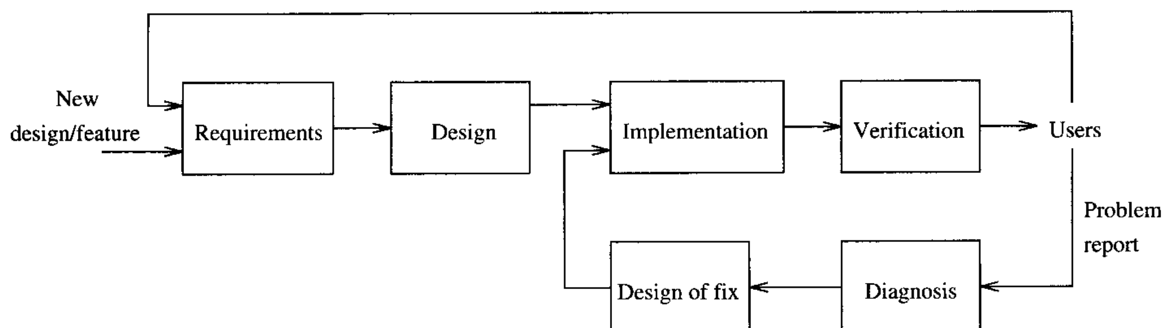


Figure 8.2 Software life cycle.

the theme of this chapter. These studies addressed the issues of fault categorization, error/failure and recovery distributions, error propagation, failure dependency, the impact of software faults on system performance and reliability, evaluation of software fault tolerance, recurrent software failures, and failure diagnosis.

8.3 Measurement Techniques

Measurement is plagued by numerous theoretical and practical difficulties. The question of what and how to measure is a difficult one. Most studies use a combination of installed and custom instrumentation. From a statistical point of view, you should collect a considerable amount of data for sound evaluations, because the accuracy of estimation is directly related to the number of samples. But the management of collected data is a nontrivial task. In an operational system, you can measure only detected errors. In modern computer systems, especially in fault-tolerant systems, failures are infrequent and, to obtain meaningful data, you should make measurements over a long period of time. Also, you should expose the measured system to a wide range of usage conditions for the results to be representative. Further, you should work with users as well as development and service organizations to collect data in the operational phase.

TABLE 8.1 Measurement-Based Studies of Software Reliability

Category	Issues	Studies
Data coalescing	Analysis of time-based tuples	[Tsao83, Hans92]
	Clustering based on type and time	[Iyer86, Lee91, Tang93a]
Software fault classification	Fault and error profile	[Lee93b, Tang92c, Hsue87] [Chil92, Thay78, Endr75]
Reliability census	Contribution of software to system reliability	[Gray90, Leve90]
Basic reliability characteristics	Error/failure bursts	[Iyer86, Hsue87, Tang93a]
	TTE/TTF/TTR distributions	[McCo79, Iyer85b, Lee93a]
Failure dependency	Hardware-related & correlated software errors	[Iyer85a, Tang92b, Lee93a]
	Two-way and multiway failure dependency	[Duga91, Lee91, Tang91]
Error propagation	First error, propagation mode, error detection	[Lee93b]
Software fault tolerance	Recovery routines	[Vela84, Hsue87]
	Process pairs	[Gray85, Gray90, Lee92, Lee93b]
Recurrences and failure diagnosis	Preventive software service	[Adam84]
	Symptom-based diagnosis of recurrences	[Lee94a]
Software reliability modeling	Performability models for error detection/recovery	[Hsue87, Hsue88]
	Two-level models for operating systems	[Lee93a, Tang93a, Lee92]
	Modeling of multiple errors	[Hsue87]
	Reliability modeling in the operational phase	[Lee94b]
Workload dependency	Workload-dependent software failure models	[Cast81, Cast82, Iyer82a] [Iyer85b, Mour87]

Establishing a sound data collection process requires ongoing cooperation between data collectors and data consumers (e.g., a practitioner). You can make two types of measurements: *on-line machine logging* and *manual reporting.* On-line logging of errors during machine operation is usually performed automatically by the operating system. Manual reports are generated by three types of data collectors: users, problem analysts, and software developers. Both types of data collection are essential for believable reliability analysis. Ideally, you should be able to cross-reference the two types of data for most incidents. Definitions and forms for data collection change as the data collection process, the software, and the hardware evolve. A sound data collection process is slow building, but it can break down easily.

The following subsections discuss issues in instrumentation and in evaluating collected data, for automatic machine logging and manual reporting.

8.3.1 On-line machine logging

Most large computer systems provide error-logging software in the operating system. This software records information on errors occurring in the operating system and its various subsystems, such as the memory, disk, and network subsystems, applications, as well as information on system events, such as reboots and shutdowns. The reports usually include information on the location, time, and type of the error, the system state at the time of the error, and error recovery (e.g., retry). The reports are stored chronologically in a permanent system file.

Figure 8.3 shows a simplified picture of on-line event logging. The collector is a system process which is in charge of event logging and event log management. Application processes can generate events when they detect abnormal conditions by means of their internal consistency checks. System processes usually generate three types of events: problems in the software components that run as system processes, problems in applications that are detected by system processes, and abnormal hardware conditions. When necessary, the human operator can intervene and collect additional data, such as the dump of a process state or the dump of a processor memory. These dumps are not usually stored in event logs because of their size, so they constitute a part of human-generated error reports discussed in the next subsection. However, they can be treated as a part of on-line machine logs in an advanced environment in which many operator tasks are programmed into the data collection module of the operating system.

Figure 8.4 shows a sample error entry extracted from a machine log from a Tandem system [TAND89]. The information in the event was decoded to make it readable. An error record consists of a header and a

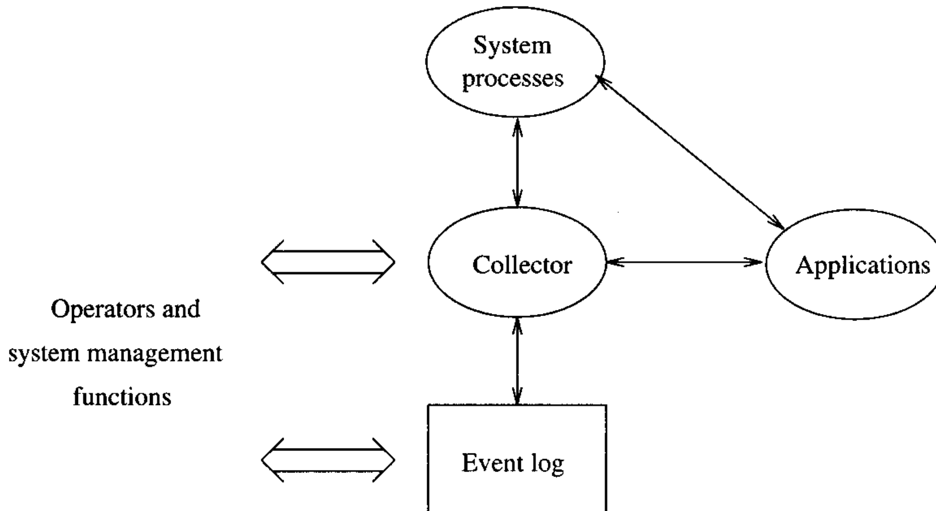


Figure 8.3 On-line event logging.

body. The header contains general information, such as the time of occurrence, the subsystem and device affected, and the type of the event. Typically, all errors have the same header format. In this example, the event reports a processor halt seemingly caused by a software fault. The body contains detailed information about the event. The format of the body differs from event to event. In Fig. 8.4, the body contains the apparent cause of the halt from an operating system perspective (halt error code) and a summary of the processor state at the time of halt.

An event in an on-line log is structured and coded in a predefined format. Issues in creating the instrumentation for automatic logging include the definition of the set of events to be reported, the signature

Time	Subsystem	Device	Event
06SEP91 09:57:00	CPU	CPU-2 SECT-1 CAB-1	CPU-Software-Halt
	CPU-Type:	3	
	Halt-error-code:	%4040	
	OS-Type:	0	
	P-register:	%60544	
	E-register:	%3407	
	L-register:	%7250	
	Current-space-id:	%147	
	Coldload-address:	%351	
	Current-PCB-address:	%107200	
	PCB-base-address:	%100100	
	DDT-status:	%10,%0,%0,%343,%120	
	DDT-error-bits:	%0,%0,%0,%0,%2,% 0,%0,%0,%0,%0,%0,%210	
	Register-file:	%%4040,%0,%4317,%1 ,%0,%57,%1360,%12742,%0,%170000 %100,%0,%177440,%53,%20,%1	

Figure 8.4 Sample error entry in error log.

format of each event, and the meaning of each field in an event. These issues are usually addressed at the system design stage, and the resulting on-line error logging is an operating system function. Related issues are the reporting mechanism between the operating system and other subsystems and the event management within the operating system. Such instrumentation should consider the possibility of introducing new events as subsystems and situations are added. Clearly, the meaning of each event is valid only within a system. You should consider the differences in hardware/software error detection and logging mechanisms when evaluating the logs from different types of systems.

The main advantage of on-line automatic logging is its ability to record a large amount of information about transient errors and on-line error recovery, which cannot be done manually. It records nearly 100 percent of key events and provides accurate timing information. However, there are several challenges in evaluating on-line logs.

First, on-line error logs do not usually provide information on underlying faults and off-line diagnosis. Also, under some crash scenarios, the system may fail too quickly for any error messages to be recorded. Therefore you should supplement machine logs with manual (human-generated) reports. Second, modern computer systems are reliable, and you should make a long period of measurement (often on a number of systems) to conduct a meaningful analysis. The size of the data can be huge. You should develop software tools to manage the data and to automate basic analysis steps. Third, the meaning of a record and the format of an event in a log can differ between versions of the operating system and between machine models. This is natural because the software and hardware of a system evolve. Thus, you should update software tools to ensure against such discrepancies.

8.3.2 Manual reporting

Manual reporting is initiated by problems found in the field (Fig. 8.2). A problem could be simple in nature, such as a misunderstanding of software features or a minor cosmetic error, or it could be as severe as a system crash or loss of data. Initially, a report contains information provided by the user, such as the time of occurrence, severity, system identification, and a description of the problem. In the case of a system crash or loss of data, additional information (e.g., a processor memory dump) is also provided. As the reported problem is diagnosed and fixed by analysts and developers, the log of all diagnostic actions, analysis history, and information on the underlying faults, failure symptoms, and fixes are appended to the report. Such information is difficult to describe using a fixed format. As a result, a manual report of a software

problem is mainly a collection of textual descriptions. Only the header has a fixed format common in all reports. Another type of report, called an *operator log,* is generated by the system operator. An operator log contains information on system crashes, failure diagnosis, and hardware and software updates.

Figure 8.5 shows a sample manual report extracted from the Tandem Product Report (TPR) database [TAND85]. A TPR is used to report all problems, questions, and requests for enhancements by customers or Tandem employees concerning any Tandem product. The figure shows mainly the header, which provides fixed fields for the information such as the date, problem type, urgency, customer and system identifications, and brief problem description. The body of a TPR is a textual description of all actions taken by Tandem analysts in diagnosing and fixing the problem. If a TPR reports a software failure, the body of the TPR also includes the log of memory dump analyses performed by analysts.

Software error reports contain detailed information about the underlying faults, symptoms, and fixes. As a result, you can use such reports to address many software reliability issues. There are two major challenges in evaluating manual reports. First, underreporting can be significant. It is estimated that the majority of processor software failures in Tandem systems are not reported [Lee93b, Gray90]. Ideally, a cross-referencing between on-line logs, manual reports, and operator logs should be possible. Second, since they are textual reports generated by humans, you cannot analyze them by automatic tools. Usually, you should reorganize the raw data into a structured database. This involves data categorization, that is, generating categories and counting instances for each category. This in turn requires understanding the details of problems long after their cases are closed, when important information may no longer be available. Such reorganization can

Tandem Product Report

TPR number: 91-01-03 17:50 Severity: 2

Product Name: GUARDIAN Kernel Origination: ABC Financial Inc.
Classification: software problem 777 Lawrence Street
Chicago, IL 60661
Date Received: 91-01-03 14:54
Date Returned: 91-01-10 10:49 System Number: 0056983

Accompanying Information: dump file location \ABC.prs.jan031750.*

Problem Description: Halts on CPUs 4 and 5.
Process $ABC runs in CPU 4 backed up in CPU 5

- -

Response:
All actions including dump analyses taken by Tandem analysts to diagnose the problem.

Figure 8.5 Human-generated software error report.

be a serious hurdle, consuming most of the evaluation effort. You can resolve this problem by generating categories before collecting data. You can provide additional space for some free text to collect information that is specific to a failure.

Table 8.2 shows an example of data categorization to collect and analyze reasons for code changes [Basi84a]. With the categories, data collectors (developers, in this case) can just mark an appropriate category on a code update. This is the most efficient and accurate way of collecting data, and it allows you to analyze the data automatically (i.e., using programs) later. In the example, the reason for collecting the information would be to take action to minimize the number of code changes in the future. Unfortunately, category generation is an imprecise science. It is usually difficult to keep categories orthogonal. That is, categories tend to overlap. Also, different people can interpret a category differently. In practice, you can overcome these problems by using a small number of well-defined categories (see Chap. 9). You should generate categories for each question to answer. Studies have shown that, with a well-defined form, it takes a minimal amount of time for human collectors to fill out the form when a case is closed.

8.4 Preliminary Analysis of Data

This section discusses preprocessing of data, fault and error classification, error propagation, and distribution identification. Such analyses investigate basic software reliability characteristics.

8.4.1 Data processing

Usually, field failure data contain a large amount of redundant and irrelevant information in various formats. Thus, you should preprocess data to extract necessary information and to put it into a database for subsequent analyses. Preprocessing varies with the types of data.

TABLE 8.2 Sample Category Generation

Issue	Category
Type of change	Error correction
	Planned enhancement
	Implementation of requirements change
	Improvement of clarity, maintainability, or documentation
	Improvement of user services
	Insertion/deletion of debug code
	Optimization of time/space/accuracy
	Adaptation to environment change
	Other

8.4.1.1 Preprocessing of automatically generated error logs. Information in error logs is usually coded and structured because it is generated automatically by the operating system. Details of preprocessing are machine-dependent because of the differences in error detection and logging mechanisms and semantics among the systems. You usually perform two major types of processing other than reformatting: *data extraction* and *data coalescing*. Data extraction means that you select the events and fields that are necessary for the analysis. Data coalescing is necessary, because a single fault in the system can result in many repeated error reports in a short period of time. To ensure that the subsequent analyses will not be distorted by these repeated reports, you should coalesce entries that correspond to the same problem into a single event.

A commonly used data-coalescing algorithm [Iyer82a] is merging all error entries of the same error type that occur within a ΔT interval of each other into a tuple. The algorithm is as follows:

```
IF <error type> = <type of previous error>
   AND <time away from previous error> ≤ ΔT
THEN <put error into the tuple being built>
ELSE <start a new tuple>
```

A tuple reflects the occurrence of one or more errors of the same type that occur in rapid succession. It can be represented by a record containing information such as the number of entries in the tuple and the time duration of the tuple.

You can make two kinds of mistakes in data coalescing: *collision* and *truncation* [Hans92]. A collision occurs when the detection of errors caused by two faults are close enough in time (within ΔT) such that they are combined into a tuple. A truncation occurs when the time between two errors caused by a single fault is greater than ΔT. In this case, the two reports are split into different tuples. If ΔT is large, collisions are likely to occur. If ΔT is small, truncations are likely to occur. You can determine the value of time-interval threshold based on data. Collision is not a big problem if you use the error type and device information in data coalescing as shown in the above coalescing algorithm. Truncation is not considered to be a problem [Hans92] because there are techniques available to deal with truncations [Iyer90, Lin90]. These techniques have been used for fault diagnosis and failure prediction.

8.4.1.2 Preprocessing of human-generated problem reports. Some information in manual reports, such as a header that includes the date, severity, and product and system identifications, is structured. The format of the rest of the report depends on the data collection process. As

explained in Sec. 8.3, you usually perform the preprocessing in three steps: (1) understanding the situations described in the reports, (2) generating categories, and (3) counting the instances of each category and constructing a database. If data categorization is done in advance and the data is collected accordingly, you may be able to skip the first and second steps.

8.4.2 Fault and error classification

Faults and errors identified from a software system can provide clues of how to fine-tune the software development environment and how to improve error detection and recovery. Fault and error categorization is frequently used in addressing such issues. Most studies have addressed the issues by using faults found during the development phase [Thay78, Endr75, Basi84a]. However, fault and error profiles of operational software are just as informative and can be quite different from those of the software in its development phase because of the differences in the operational environment and the maturity of software. Therefore, to improve software quality, it is important to investigate software fault and error profiles in the field. Here we introduce fault and error profiles obtained using field data collected from Tandem GUARDIAN, IBM MVS, and VAX VMS operating systems [Lee93b, Hsue87, Tang92c].

8.4.2.1 GUARDIAN. Table 8.3 shows the results of a fault classification using 153 Tandem Product Reports (TPRs) that contain logs of processor dump analyses of software failures performed by analysts. The table shows both the number of TPRs and the number of unique faults. The differences between the two represent multiple failures caused by the same fault. From Table 8.3, you can see what kinds of faults the developers introduced. "Incorrect computation" refers to an arithmetic overflow or the use of an incorrect arithmetic function (e.g., use of a signed arithmetic function instead of an unsigned one). "Data fault" refers to the use of an incorrect constant or variable. "Data definition fault" refers to a fault in declaring data or in defining a data structure. "Missing operation" refers to an omission of a few lines of source code. "Side effect of code update" occurs when not all dependencies between software modules are considered when updating software. "Unexpected situation" refers to cases in which the software designers did not anticipate a legitimate operational scenario, and consequently the software did not handle the situation correctly. You can see from the table that "Missing operation" and "Unexpected situation" are the most common types of software faults in Tandem systems. Additional code inspection and testing efforts can be directed for identifying such faults.

TABLE 8.3 Software Fault Categorization for GUARDIAN

Fault Category	No. faults		No. TPRs	
Incorrect computation		3		3
Data fault		12		21
Data definition fault		3		7
Missing operation:		20		27
—Uninitialized pointers	(6)		(7)	
—Uninitialized nonpointer variables	(4)		(6)	
—Not updating data structure on the occurrence of certain events	(6)		(9)	
—Not telling other processes about the occurrence of certain events	(4)		(5)	
Side effect of code update		4		5
Unexpected situation:		29		46
—Race/timing problem	(14)		(18)	
—Errors with no defined error-handling procedures	(4)		(8)	
—Incorrect parameters or invalid calls from user processes	(3)		(7)	
—Not providing routines to handle legitimate but rare operational scenarios	(8)		(13)	
Microcode defect		4		8
Others (cause does not fit any of the above class)		10		12
Unable to classify due to insufficient information		15		24
All		100		153

A software failure caused by a newly found fault is referred to as a *first occurrence;* a software failure caused by a previously reported fault is referred to as a *recurrence.* The 153 TPRs whose software causes were identified occurred due to 100 unique faults (Table 8.3). Out of the 100 software faults observed during the measured time window, 57 faults were diagnosed before the time window (i.e., were recurrences) and 43 were newly identified during the time window (i.e., were first occurrences). That is, about 72 percent (110 out of 153) of the TPRs reported recurrences of previously reported software faults. The issue of recurrence is discussed further in Sec. 8.5.4.

8.4.2.2 MVS. In MVS, software error data on the type of detection (hardware and software) and recovery are logged by the system onto a data set called SYS1.LOGREC. Each error record in the LOGREC data contains bits describing the type of error, its severity, and the results of hardware and software attempts to recover from the problem. The general software error status indicators are TYPE (of detection), EVENT (causing the detection), and ERRCODE (code of symptom of the error). Based on the ERRCODE information provided by the system, eight classes of errors, which reflect commonly encountered problems, were defined [Hsue87]:

1. *Control* (CTRL) indicates the invalid use of control statements or invalid supervisor calls.
2. *Deadlock* (DLCK) indicates endless loops, wait states, or violation of system- or user-defined time limits.

TABLE 8.4 Software Error Classification for MVS

(Measurement period: 12 months)

Error type	Frequency	Percent
Control	213	7.72
Deadlock	23	0.84
I/O & data management	1448	52.50
Program exception	65	2.43
Storage exception	149	5.40
Storage management	313	11.35
Others	66	2.32
Multiple error	481	17.44
All	2758	100.00

3. *I/O and data management* (I/O) indicates a problem occurred during I/O management or during the creation and processing of data sets.
4. *Storage management* (SM) indicates an error in the storage allocation/deallocation process or in virtual memory mapping.
5. *Storage exception* (SE) indicates addressing of nonexistent or inaccessible memory locations.
6. *Programming exception* (PE) indicates a program error other than a storage exception.
7. *Others* (OTHR) indicates errors which do not fit any of the above categories.
8. *Multiple errors or error bursts* (MULT) indicates error bursts consisting of different types (listed above) of errors.

Table 8.4 lists the frequencies of different types of software errors defined above. You can see that more than one half (52.5 percent) of software errors are I/O and data management errors and another 11.4 percent of the errors are storage management errors. This result is probably related to the fact that a major feature of MVS is multiple virtual storage organization. Also, I/O and data management is a high-volume activity critical to the proper operation of the system. It is therefore expected that their contributions are significant. You can also see that a significant percentage of errors are multiple errors, indicating that error detection and recovery need to take multiple errors into account (to be discussed further in Sec. 8.6.4).

8.4.2.3 VMS. Software errors in a VAXcluster system are identified from *bugcheck* reports in the error log files. All software-detected errors were extracted from bugcheck reports and divided into four types in [Tang92c]:

TABLE 8.5 Software Error Classification for VMS

(Measurement period: 10 months for VAX1 and 27 months for VAX2)

Error type	Frequency (VAX1)	Frequency (VAX2)	Fraction (%), combined
Control	71	26	50.0
Memory	8	4	6.2
I/O	16	44	30.9
Others	1	24	12.9
All	96	98	100

1. *Control.* Problems involving program flow control or synchronization. For example, "Unexpected system service exception," "Exception while above ASTDEL (Asynchronous System Traps DELivery) or on interrupt stack," and "Spinlock(s) of higher rank already owned by CPU."
2. *Memory.* Problems referring to memory management or usage. For example, "Bad memory deallocation request size or address," "Double deallocation of memory block," "Pagefault with IPL (Interrupt Priority Level) too high," and "Kernel stack not valid."
3. *I/O.* Inconsistent conditions detected by I/O management routines. For example, "Inconsistent I/O data base," "RMS (Record Management Service) has detected an invalid condition," "Fatal error detected by VAX port driver," and "Invalid lock identification."
4. *Others.* Other software-detected problems. For example, "Machine check while in kernel mode," "Asynchronous write memory failure," and "Software state not saved during powerfail." These are actually not software-related errors although their statistics are included.

Table 8.5 shows the frequency for each type of software-detected error for the two measured VAXcluster systems. You can see that nearly 13 percent of software-detected errors are type "Others," and almost all of them belong to VAX2. The VAX2 data show that most of these errors are "machine check" (i.e., CPU error). The VAX1 error logs do not include CPU errors in the bugcheck category. A careful study of the VAX error logs and discussions with field engineers indicate that different VAX machine models may report the same type of error (in this case, CPU error) to different classes. Thus, it is necessary to distinguish these errors in the error classification. Most "Others" errors are judged to be nonsoftware problems.

8.4.3 Error propagation

Given that a complete elimination of software faults in a large, continually evolving software system is difficult, it is important that the soft-

ware handles the effects of software faults efficiently. Such a design requires understanding the effects of software faults and establishing efficient software fault models. While efficient models for hardware faults exist, the issue of software fault models is open.

You can build software fault models from two perspectives: software engineering and software fault tolerance. Examples of software fault models built from the software engineering perspective are those resulting from software fault categorization. You can use such models for fine-tuning the software development environment and for avoiding or eliminating software faults. Software fault models built from the software fault tolerance perspective are those based on a knowledge of faults, the effects of software faults (i.e., errors), error propagation, or a combination of these. You can use such models for designing efficient error detection, diagnosis, and recovery strategies. This subsection discusses a re-creation of error propagation using the 153 TPRs used to create Table 8.3 in Sec. 8.4.2, to build a model from the software fault tolerance perspective.

The term *first error* is defined as the immediate effect of a software fault on the processor state when the fault is exercised. In other words, the first error of a software fault refers to the first program variable that acquires an incorrect value because of the fault. The first errors identified from the 153 TPRs were classified into the five categories [Lee93b]:

1. *Single address error.* An incorrect address word is developed.
2. *Single nonaddress error.* An incorrect nonaddress value is developed. Instances in this category are further divided into four subclasses: incorrect field size, incorrect index, incorrect flag, and the rest.
3. *Multiple errors.* Multiple errors are generated at once. Instances in this category are further divided into two subclasses: (1) random corruption in a memory area without regard to the data structure (e.g., a corruption caused by a stack area overlap or a missing initialization of a memory area) and (2) multiple regular errors in data structure (e.g., memory management tables become inconsistent due to a partial update, or a request buffer is overwritten by another request).
4. *Others.* The first error does not fit any of the above categories (e.g., an invalid request caused by a race condition).
5. *Unable to classify.* The first error cannot be identified due to insufficient information in the TPRs.

The propagation characteristics of first errors were classified into three groups: *no propagation, further corruption,* and *quick detection.*

No propagation refers to cases in which there is no possibility of error propagation, i.e., the first error is certain to be detected on the first access. Further corruption refers to error propagation across processes and the generation of more errors. Quick detection lies between the above two propagation modes. In this situation, there is no guarantee that there will be no propagation. The problem is detected quickly, after the first error is accessed for the first time, while the task that made the first access is executed.

Figure 8.6 shows an overall picture of error propagation, from underlying software faults to problem detection. A circle or a rectangle represents a category, and the numbers inside it represent the number of TPRs in that category and its percentage of the 153 TPRs. An arrow represents a transition, and the associated number represents a branching probability from the source state. For example, you can see that data faults account for 14 percent of the faults, and if a fault in this category is exercised, there is a 24 percent chance that an incorrect address will be generated. Figure 8.6 captures all major error propagation paths that must be eliminated. You can observe from the figure that address errors are difficult to handle with consistency checks. The data show no instances in which "Single address" error is guaranteed to be detected on the first access.

In Figure 8.6, "No propagation" is a desired state, because it does not threaten the integrity of the data in the system. It is a significant state

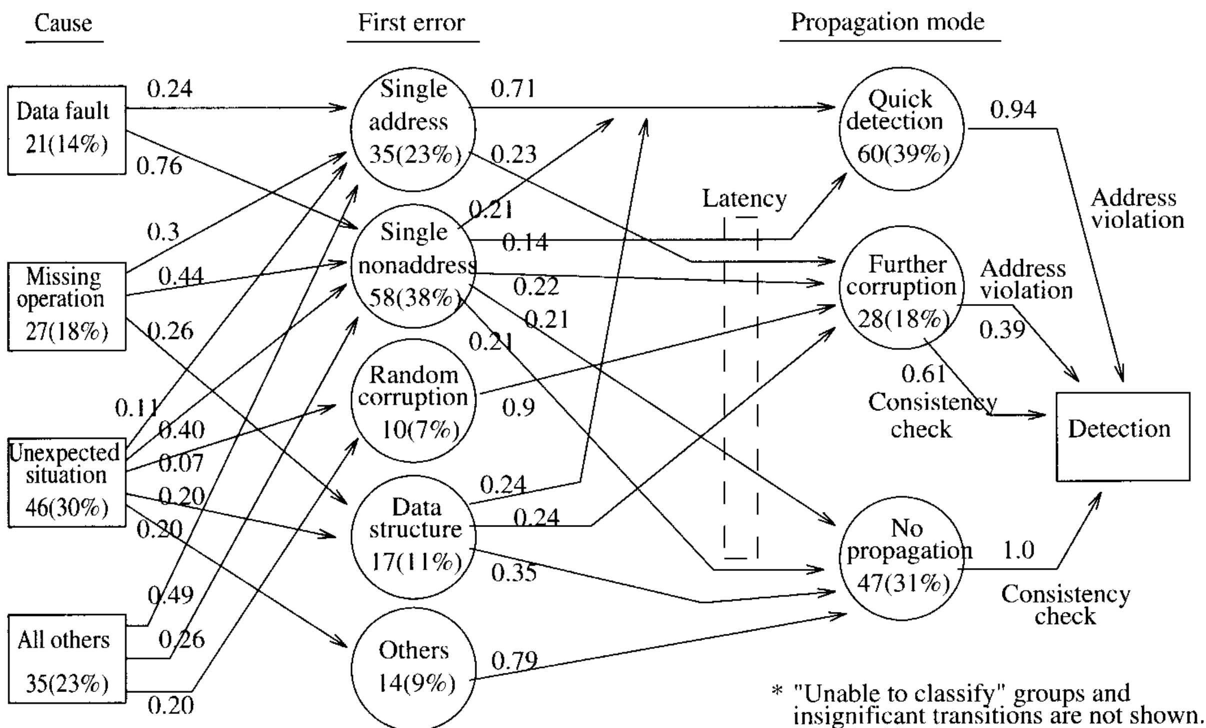


Figure 8.6 Error propagation model.

in the measured system because of the use of redundant data structures and consistency checks. You can see that all instances of "No propagation" are detected by consistency checks. In 94 percent of the instances of "Quick detection," problems are detected due to address violations; the rest are detected by consistency checks.

"Further corruption" is a dangerous state, in that error propagation can occur recursively and multiple errors are generated until the problem is detected. Note that some of the errors may break the fault-containment boundary assumed for on-line recovery and thus cause another problem later. Any process that accesses corrupted data can potentially assert a halt, and thus a single fault can cause a variety of failure symptoms, which may complicate the diagnosis.

8.4.4 Error and recovery distributions

Probably the most basic software reliability characteristics are time to failure/error (TTF/TTE) and time to repair (TTR) distributions. This subsection discusses error/failure frequency and empirical distributions built from data.

8.4.4.1 Error/failure frequency. It is often convenient to count the numbers of different types of errors and failures during the measurement period. You can make an easy comparison of error types and can also identify reliability bottlenecks using these counts. Table 8.6 shows the error/failure statistics for a VAXcluster system [Tang93a]. In the table, I/O errors include disk, tape, and network errors. Machine errors include CPU and memory errors. Software errors are software-related errors. Recovery probability is the probability that an error does not cause a machine crash.

You can identify two bottlenecks from the table. First, although software errors constitute only a small portion of all errors (0.3 percent), they result in significant failures (25 percent). This is because software errors have a very low recovery probability (0.1). Second, the major error category is I/O errors (93 percent), i.e., errors from shared

TABLE 8.6 Error/Failure Statistics for the VAXcluster

	Error		Failure		
Category	Frequency	Fraction (%)	Frequency	Fraction (%)	Recovery probability
I/O	25807	92.9	105	42.9	0.996
Machine	1721	6.2	5	2.0	0.970
Software	69	0.3	62	25.3	0.101
Unknown	191	0.7	73	29.8	0.618
All	27788	100.0	245	100.0	0.99

resources. This category of error has a very high recovery probability (0.996). However, these errors still result in nearly 43 percent of all failures. This result indicates that, although the system is generally robust with respect to I/O errors, the shared resources still constitute a major reliability bottleneck due to the sheer number of errors. Improving such a system may require using an ultrareliable network and disk system to reduce the raw error rate, not just providing high recoverability.

8.4.4.2 Error distributions. A realistic, analytical form of TTE distribution is essential in modeling and evaluating software reliability. You can obtain such distributions using the procedure described in App. B. You can sometimes satisfactorily represent a raw distribution by multiple distributions, which are chosen based on data, prior knowledge, and intuition. You can gain insights into different aspects of the data from each of these fits. Often, for simplicity or due to lack of information, the TTE is assumed to be exponentially distributed [Arla90, Lapr84].

Figure 8.7 shows the empirical TTE or TTF distributions fitted to analytic functions for Tandem GUARDIAN, DEC VAX VMS, and IBM MVS operating systems [Lee93a]. Here, a failure means a processor or machine failure, not a system failure. None of these distributions fit simple exponential functions. The fitting was tested using the Kolmogorov-Smirnov or chi-square test at a 0.05 significance level. This

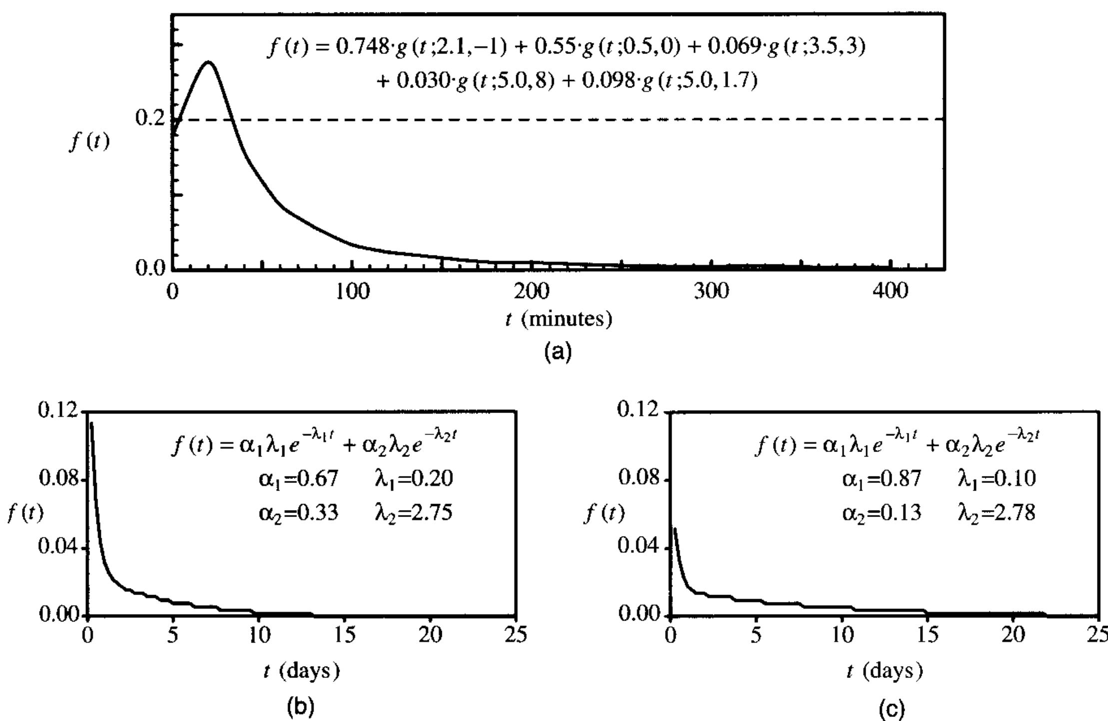


Figure 8.7 Empirical software TTE/TTF distributions. (*a*) IBM MVS software TTE distribution; (*b*) VAXcluster software TTE distribution; (*c*) Tandem software TTF distribution.

result conforms to the previous measurements on IBM [Iyer85a] and DEC machines [Cast81, McCo79]. Several reasons for this nonexponential behavior, including the impact of workload, were documented in [Cast81].

The two-phase hyperexponential distribution provided satisfactory fits for the VAXcluster software TTE and Tandem software TTF distributions. An attempt to fit the MVS TTE distribution to a phase-type exponential distribution led to a large number of stages. As a result, the multistage gamma distribution was used. It was found that a five-stage gamma distribution provided a satisfactory fit.

Figure 8.7*b* and *c* shows that the measured software TTE and TTF distributions can be modeled as a probabilistic combination of two exponential random variables, indicating that there are two dominant error modes. The higher error rate, λ_2, with weight α_2, captures both the error bursts on a single instance of an operating system and concurrent errors on multiple instances of an operating system (Sec. 8.5.1). The lower error rate, λ_1, with weight α_1, captures regular errors and provides an interburst error rate.

These error bursts may be repeated occurrences of the same software problem or multiple effects of an intermittent hardware fault on the software. Actually, software error bursts have been observed in laboratory experiments reported in [Bish88]. The study showed that, if the input sequences of the software under investigation are correlated (rather than being independent), one can expect more bunching of failures than those predicted using a constant failure rate assumption. In an operating system, input sequences (user requests) are highly likely to be correlated. Hence, a defect area can be triggered repeatedly.

8.4.4.3 Recovery distributions. Figure 8.8*a* plots the spline-fit for the TTR distribution of multiple software errors in the MVS system [Lee93a]. A multiple software error is an error burst consisting of different types of software errors. The TTR distribution for multiple software errors is presented because these errors have longer recovery times than other software errors and are more typical in terms of recovery process (Table 8.14 in Sec. 8.6.3). A three-phase hyperexponential function could be used to approximate the distribution, suggesting a multiple mode recovery process. Because most MVS software errors do not lead to system failures, the TTR for multiple errors is short, although these errors take the longest time to recover of all software errors.

Figure 8.8*b* and *c* plots the empirical software TTR distributions for the VAXcluster and Tandem systems. Because of their peculiar shapes, the raw distributions are provided. Since each system has different error semantics, recovery procedures, and maintenance environments, you cannot compare the measured systems in terms of TTR distribution. In the VAXcluster (Fig. 8.8*b*), you can see that most of the TTR

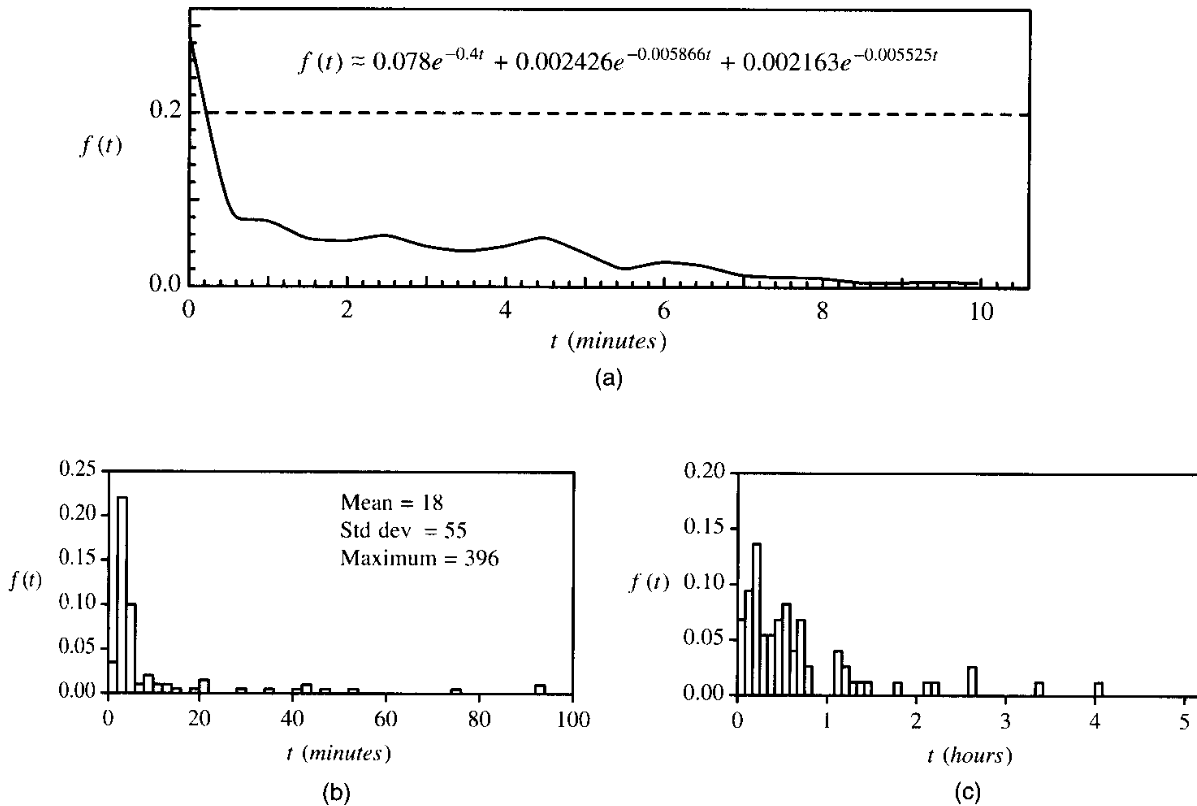


Figure 8.8 Empirical software TTR distributions. (*a*) MVS multiple software error TTR distribution; (*b*) VAXcluster software TTR distribution; (*c*) Tandem software TTR distribution.

instances (85 percent) are less than 15 minutes. This is attributed to those errors recovered by on-line recovery or automatic reboot without shutdown repair. However, some TTR instances last as long as several hours (the maximum is about 6.6 hours). These failures are, in our experience, probably due to a combination of software and hardware problems. Since the Tandem system does not allow an automatic recovery from a halt and all events considered are processor halts due to software, its TTR distribution (Fig. 8.8*c*) reflects the time to collect failure data and to reload and restart by the operator.

Typically, analytical models assume exponential or constant recovery times. You can see from Fig. 8.8 that this does not apply universally. None of the three TTR distributions is a simple exponential. For the MVS system, since the recovery is usually quick, a constant recovery time assumption may be suitable. For the VAXcluster and Tandem systems, neither exponential nor constant recovery time can be assumed. You should use more complex multimode functions to model these TTR distributions.

8.5 Detailed Analysis of Data

After preliminary analysis of data, you can perform a series of analyses that evaluate features specific to the measured software system and

data. You can gain insights into the types of analysis to be performed additionally from the results of preliminary analysis. You can also perform a detailed analysis based on specific analysis goals set in advance. This section discusses the analysis of failure dependency, hardware-related software failures, evaluation of software fault tolerance due to the use of process pairs and recovery routines, and the issue of recurrence.

8.5.1 Dependency analysis

Many underlying dependencies can exist among measured parameters and components. Examples are the dependency between workload and failure rate and the dependency or correlation among failures on different system components. Failure dependency is a special concern in fault-tolerant systems and highly parallel systems. Nonetheless, few measurement-based studies have addressed this issue. While you can use analytic methods, such as Markov modeling, to represent the failure dependencies identified, you can identify the types of dependencies that exist in actual systems and the range of realistic dependency parameters based only on field data. Understanding and quantifying such dependencies is important for developing realistic models and hence better designs. This subsection introduces a real example of correlated software errors and two studies of failure dependency based on real data: (1) an analysis of the two-way dependency between errors on two different machines in a VAXcluster system [Tang93a] and (2) an analysis of multiway dependency among failures on multiple processors in a Tandem fault-tolerant system [Lee91].

8.5.1.1 Correlated software errors. When multiple instances of a software system interact with each other in a multicomputer environment, you should consider the issue of correlated failures. Several studies ([Tang90, Wein90, Lee91]) found that significant correlated processor failures exist in multicomputer systems. To understand how correlated software failures occur, we will examine a real case in detail.

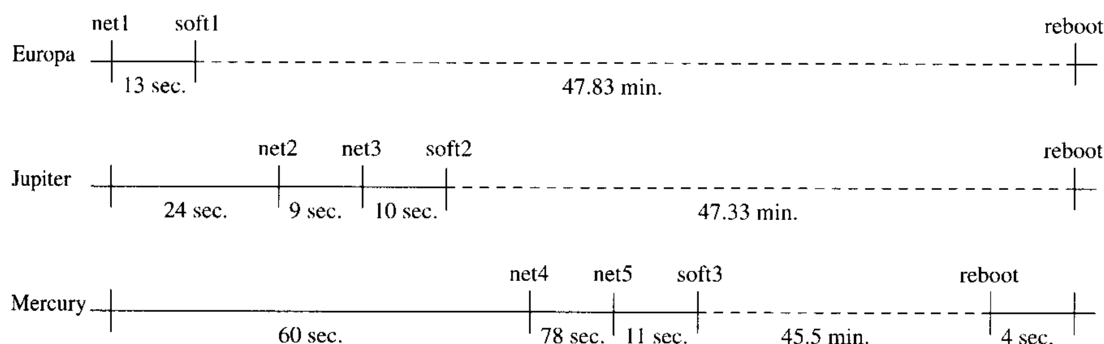
Figure 8.9 shows a scenario of correlated software failures observed in a seven-machine VAXcluster [Lee93a]. In the figure, Europa, Jupiter, and Mercury are machine names in the VAXcluster. A dashed line represents that the corresponding machine is in a failure state. Initially, a network error (net1) was reported from the CI (computer interconnect) port on Europa. This resulted in a software failure (soft1) 13 seconds later. Twenty-four seconds after the first network error (net1), additional network errors (net2,net3) were reported on the second machine (Jupiter), which were followed by a software failure (soft2). The error sequence on Jupiter was repeated (net4,net5,soft3) on

the third machine (Mercury). The three machines experienced software failures concurrently for 45.5 minutes. All three software failures occurred shortly after network errors occurred, so they were probably network-error-related.

Note that the above scenario is a multiple-component-failure situation. A substantial amount of effort has been directed at developing general system design principles against correlated failures. Still, correlated failures exist due to design holes and unmodeled faults. Generally, correlated failures can stress recovery and break the protection provided by the fault tolerance.

8.5.1.2 Two-way dependency. The first step in a dependency analysis is to build a data matrix based on the measured data. Assume that there are n components in the measured system and that the measured period is divided into m equal intervals of Δt (e.g., 30 minutes). Then you can construct an $m \times n$ data matrix such that the element (i, j) of this matrix has a value of 1 if component j experiences an error or a failure during the ith time interval; otherwise, it has a value of 0. Alternatively, you can define the value of the element (i, j) of the matrix as the number of errors or failures occurred in component j during the ith interval. Note that the jth column of the matrix represents the sample error or failure history of component j, while the ith row of the matrix represents the state of the components in the ith time interval.

You can calculate correlation coefficients based on the data matrix. Each time, you pick up two columns (X_i and X_j) to calculate cor(X_i,X_j). Table 8.7 lists the average correlation coefficients of the 21 pairs of machines in a seven-machine VAXcluster for different types of errors and failure [Tang93a]. The table also lists the recovery probability for each error type. You can see that disk errors have the strongest corre-



Note: soft1, soft2, soft3 — Exception while above asynchronous system traps delivery or on interrupt stack.
net1, net3, net5 — Port will be restarted. net2, net4 — Virtual circuit timeout.

Figure 8.9 A scenario of correlated software failures.

lation. This is because errors in the disk subsystem often affect multiple machines because of the sharing of disks. For similar reasons, network errors are strongly correlated across machines. While software error correlation across the machines is low, software failure correlation across them is significant because of the low recovery probability from a software error. This result is significant because even a small failure correlation can have a significant impact on system availability.

8.5.1.3 Multiway dependency. The limitation of correlation analysis is that the correlation coefficient can quantify a dependency between two variables only. However, dependencies may exist within a group of more than two variables or even among all variables. For example, in a distributed system, a disk crash can cause failures on those machines whose operations depend on a set of critical data on the disk, resulting in multiway failure dependency in these machines.

Multivariate analysis techniques allow you to analyze multiway failure dependency. Principal component analysis, factor analysis, and cluster analysis were used to identify the multiway failure dependency (see App. B) in [Lee91]. Table 8.8 shows the results of factor analysis using processor halt data collected from an eight-processor Tandem system.

According to [Dill84], factor loadings greater than 0.5 are usually considered significant. However, in reliability analysis, factor loadings

TABLE 8.7 Average Correlation Coefficients for VAXcluster Errors

	Error						Failure
	All	CPU	Memory	Disk	Network	Software	All
Correlation coefficient	0.62	0.03	0.01	0.78	0.70	0.02	0.06
Recovery probability	0.99	0.97	1.00	0.99	0.99	0.08	—

TABLE 8.8 Factor Pattern of Processor Halts in a Tandem System

Processor	Common factor 1	Common factor 2	Common factor 3	Common factor 4	Communality
1	0.997	−0.004	−0.069	0.023	1.00
2	0.000	0.000	0.000	0.000	0.00
3	0.061	0.012	0.853	−0.133	0.75
4	0.001	0.999	−0.011	0.021	1.00
5	0.982	−0.000	0.188	−0.018	1.00
6	−0.001	0.447	−0.005	0.009	0.20
7	0.047	−0.002	0.862	0.506	1.00
8	−0.007	0.762	0.090	0.641	1.00
Var.	1.965	1.781	1.519	0.685	
Var. %	24.6	22.3	19.0	8.6	

lower than 0.5 can be significant because even a small correlation can have a significant impact on system reliability. The results of factor analysis (Table 8.8) show that there are four common factors. You can see that, for example, common factor 1 captures the dependency between processors 1 and 5 and accounts for 24.6 percent of the total variance. Common factor 2 captures the multiway dependency among processors 4, 6, and 8, although the contribution of processor 6 is smaller (0.447^2, that is, 20 percent of its variance is explained by this factor). Common factor 2 explains 22.3 percent of the total variance. You can identify hidden failure dependencies and model the impact of design improvements on system reliability from such an analysis. The development of techniques to model such multiway dependencies efficiently is an area of future work.

8.5.2 Hardware-related software errors

When software is running on hardware platforms, interactions between hardware and software occur. Such interactions and their effects on system reliability are particularly difficult to comprehend. This is further compounded by the lack of real data. Results based on actual measurements and experiments are essential for developing a clear understanding of the problem.

The operating system's handling of software errors related to hardware was first studied using on-line event logs in [Iyer85a]. Such errors are described as *hardware-related software errors* (or HW/SW errors). More precisely, if a software error (failure) occurs in close proximity (within a minute) to a hardware error, it is called a hardware-related software (HW/SW) error (failure). You can explain hardware-related software errors in several ways. For instance, a hardware error, such as a flipped memory bit, may change the software conditions, resulting in a software error. Therefore, even though the error is reported as a software error, it is actually caused by faulty hardware. Another possibility is that software may fail to handle an unexpected hardware status, such as an unusual but legitimate condition in the network communication. This is a software design flaw. Sometimes, both the hardware error and the software error are symptoms of another, unidentified problem.

Table 8.9 shows the frequency and percentage of HW/SW errors/failures (among all software errors/failures) measured from an IBM 3081 system running MVS [Iyer85b] and two VAXclusters [Tang92b]. In the IBM system, approximately 33 percent of all observed software failures are hardware-related. HW/SW errors are found to have large error-handling times (high recovery overhead). The MVS data show that the system failure probability for HW/SW errors is close to three times

that for software errors in general and that the operating system is seldom able to diagnose that a software error is hardware-related. The VAXcluster data show that most hardware errors involved in HW/SW errors are network errors (75 percent). This is probably because processes rely heavily on communications through the network in the multicomputer system.

8.5.3 Evaluation of software fault tolerance

Two major approaches proposed for software fault tolerance are recovery blocks and N-version programming [Aviz84, Rand75] (see Chap. 14). Both of these approaches require multiple, independently generated versions of software. As a result, they are not easily applicable to large, continually evolving software systems due to cost constraints, although critical code sections can be protected by these techniques.

It has been observed that some techniques originally intended for hardware fault tolerance can cope with software faults [Gray85, Gray90]. Detailed evaluation of software fault tolerance achieved by the use of process pairs in the Tandem GUARDIAN operating system and recovery routines in the IBM MVS operating system has been performed [Lee95, Vela84]. Process pairs are an implementation of the checkpointing and restart technique, which is a general approach. Recovery routines are a systematic implementation of exception handling. [Lee95] showed that the use of process pairs in Tandem systems, which was originally intended for tolerating hardware faults, allows the system to tolerate about 75 percent of reported field faults in the system software that cause processor failures. The loose coupling between processors, which results in the backup execution (the processor state and the sequence of events occurring) being different from the original execution, is a major reason for the measured software fault tolerance. This result shows that there is another dimension in achieving software fault tolerance. (Refer to the cited works for further details.) Clearly, software reliability can be improved by designs exploiting such knowledge in similar environments. Recently, attempts have been made to exploit the subtle nature of some software faults to tolerate such faults in user applications using checkpointing and restart [Huan93, Wang93].

TABLE 8.9 Hardware-Related Software Errors/Failures

Category	HW/SW errors		HW/SW failures	
Measures	Frequency	Percent	Frequency	Percent
IBM/MVS	177	11.4	94	32.8
VAX/VMS	32	18.9	28	21.4

8.5.4 Recurrences

[Lee93b] showed that about 72 percent of reported field software failures in Tandem systems are recurrences (Sec. 8.4.2). Recurrences are not unique in Tandem systems. A similar situation exists in IBM systems [Adam84] and AT&T systems [Leve95]. This shows that the number of faults identified in software is not the only important factor. Recurrences can seriously degrade software reliability in the field.

Recurrences exist for several reasons. First, designing and testing a fix of a problem can take a significant amount of time. In the meantime, recurrences can occur at the same site or at other sites. Second, the installation of a fix sometimes requires a planned outage, which may force users to postpone the installation and thus cause recurrences. Third, a purported fix can fail. Finally, and probably most importantly, users who did not experience problems due to a certain fault often hesitate to install an available fix for fear that doing so will cause new problems.

This subsection discusses two issues to reduce the impact of recurrences: software service policy to minimize the number of recurrences taking the cost of service into consideration [Adam84] and automatic diagnosis of recurrences based on their symptoms [Lee94a].

8.5.4.1 Preventive software service. *Corrective service* is the process of eliminating a software fault from a user's code after the fault caused a problem to the user. *Preventive service* is the process of eliminating a software fault from a user's code when the fault has not yet caused a problem to the user. Preventive service can potentially reduce the number of recurrences, but it requires resources to prepare, distribute, and install fixes. More important, it can cause additional problems because of faults in the fixes. Then question is: what is the optimal preventive service policy?

Based on the failure and shipment data of IBM products, [Adam84] proposed a procedure to predict the number of recurrences. Table 8.10 shows a sample rediscovery matrix constructed using the procedure. Here the terms *recurrence* and *rediscovery* are used interchangeably. The rows and columns of the matrix are labeled by months counted from the time of first customer shipment. The entry (i, j) of the matrix is the number of projected rediscoveries in the user base during month i caused by faults first discovered in month j. The total of the numbers in the ith row is the total number of rediscoveries expected in month i. The numbers are projected for a hypothetical product that has steady month-to-month growth of usership, assuming that all users use the initial version of the product.

You can see from the rediscovery matrix that the number grows steadily down a column and diminishes strongly to the right across the

TABLE 8.10 Rediscovery Matrix

		Discovery month																	
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Rediscovery month	1	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	2	29	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	3	44	26	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	4	59	35	19	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	5	73	43	24	15	5	0	0	0	0	0	0	0	0	0	0	0	0	0
	6	88	52	28	17	12	4	0	0	0	0	0	0	0	0	0	0	0	0
	7	103	61	33	20	14	10	4	0	0	0	0	0	0	0	0	0	0	0
	8	117	69	38	23	16	11	8	3	0	0	0	0	0	0	0	0	0	0
	9	132	78	42	26	18	13	9	7	3	0	0	0	0	0	0	0	0	0
	10	147	87	47	29	20	14	10	8	6	2	0	0	0	0	0	0	0	0
	11	162	95	52	32	22	15	11	9	7	5	2	0	0	0	0	0	0	0
	12	176	104	57	35	24	17	12	9	7	6	5	2	0	0	0	0	0	0
	13	191	113	61	38	25	18	13	10	8	6	5	4	2	0	0	0	0	0
	14	206	121	66	41	27	19	14	11	8	7	5	4	3	1	0	0	0	0
	15	220	130	71	44	29	21	15	12	9	7	6	5	4	3	1	0	0	0
	16	235	139	76	47	31	22	16	12	10	8	6	5	4	3	3	1	0	0
	17	250	147	80	50	33	24	17	13	10	8	6	5	4	4	3	3	1	0
	18	264	156	85	52	35	25	18	14	11	9	7	6	4	4	3	3	2	1

row. The variation down the column reflects the continual entry of new users who can have problems; the decrease to the right reflects the diminishing virulence of faults found in later months. Table 8.10 shows that the large numbers all occur in the columns to the left. This indicates that you might be better off by limiting preventive software service only to a relatively small number of highly visible faults that cause problems in leftmost columns.

8.5.4.2 Automatic diagnosis of recurrences. The above mentioned study ([Adam84]) and the reasons for recurrence indicate that recurrences will continue to be a significant part of field software failures. Then the question is: how can you handle recurrences efficiently? An approach to automatically identify recurrences based on their symptoms has been proposed in [Lee94a]. The approach is based on an observation that failures caused by the same fault often share common symptoms [Lee93b]. Specifically, the study proposed the comparison of failure symptoms, such as the stack trace and code location where problems were detected, as a strategy for identifying (i.e., diagnosing) recurrences. A stack trace is the history of procedure calls made by the active process at the time of failure. It represents the software function that detected a problem.

Figure 8.10 illustrates the type of automatic diagnosis environment envisioned. The diagnosis tool is connected with many user systems by an on-line alarm system. All previously reported failure symptoms and associated information, such as underlying faults and fixes, are stored in a database. On a failure alarm, the tool accesses the system that sent the alarm, extracts the values of the common symptoms (e.g., a stack trace and a detection location), and compares them with those of previously reported faults in the database. If a match is found in the

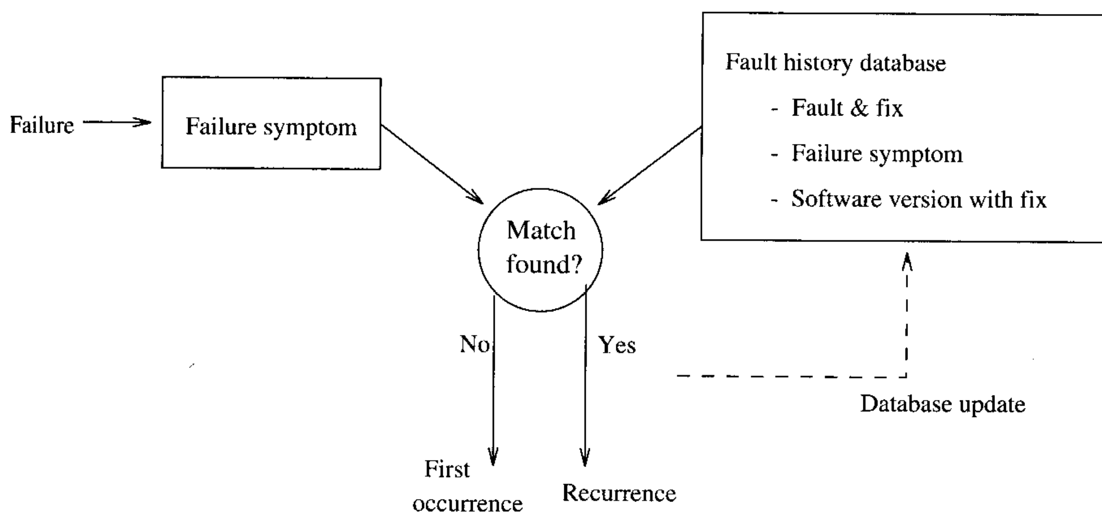


Figure 8.10 Diagnosis environment.

database, the new failure is declared a recurrence of the corresponding fault; otherwise, it is declared a first occurrence. In the case of recurrence, the tool also identifies an available fix. After the diagnosis, the database is updated with new failure data.

To apply symptom-based diagnosis of software failures, you should consider the following two extremes: a software fault can cause failures with different symptoms, and two software faults can cause failures with identical symptoms. Figure 8.11 illustrates the first extreme: two failures caused by the same software fault have different stack traces. In the figure, a circle represents a procedure call, and an arrow represents the execution within a procedure. Figure 8.11 shows a failure in which the base procedure MAINLOOP called the procedure NEXTREQ, which in turn called the procedure MONITORPRIMARY. MONITORPRIMARY called the procedure TK_PROCESS_TK_CKPT, in which a fault was exercised and a processor halt was asserted. In another failure, the same sequence was repeated, except that MAINLOOP reached MONITORPRIMARY through the procedure INITIALIZE. This calling path is also shown in the figure. Each chain of procedure calls forms a stack trace and is represented by a set of connected solid arrows in the figure. Because the software structure is modular, there can be different program paths that reach the faulty code section. Figure 8.11 shows two such paths. Each of the paths gives a distinct stack trace.

Another example of the first extreme is the case of a wide range of corruption in shared data. In this case, any software function can detect some of the errors and assert a processor halt. This would lead to widely different stack traces, problem detection locations, and error patterns.

The second extreme to consider is that different faults can cause failures with identical symptoms. A procedure typically contains multiple

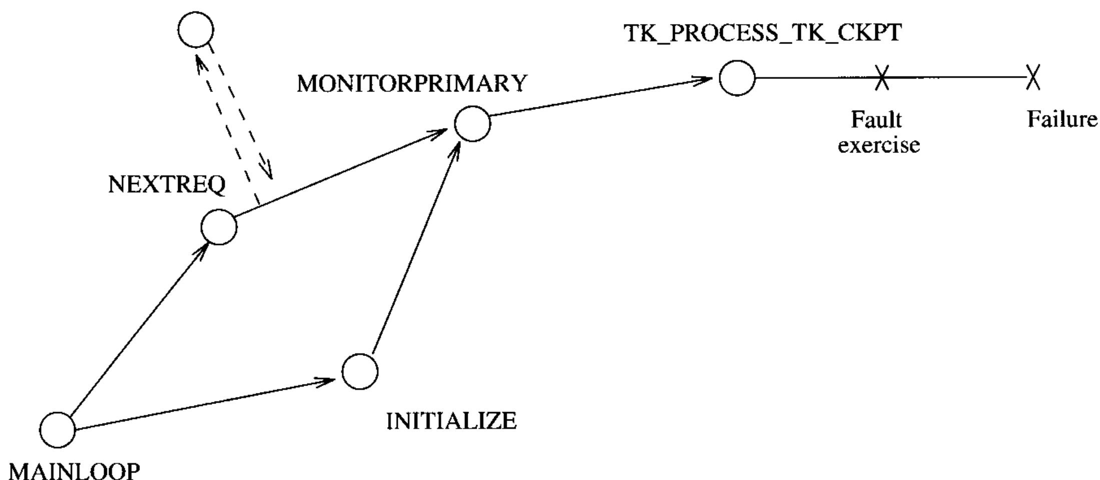


Figure 8.11 A single fault causing failures with different symptoms.

checks, and these checks test different conditions. As a result, errors caused by different faults can be detected within the same procedure, thus resulting in failures with identical stack traces.

You can clearly see that the effectiveness of a diagnosis strategy under the two extremes must be evaluated using the actual data. The proposed strategy was applied using the failure data from two Tandem system software products [Lee94a]. Then the results obtained were compared with the actual diagnosis and repair logs by analysts. Results of the comparison showed that between 75 and 95 percent of recurrences can be identified successfully by matching stack traces and problem detection locations. Less than 10 percent of faults are misdiagnosed.

The results show that the proposed automatic diagnosis of recurrences allows analysts to diagnose only one out of several software failures (i.e., primarily the failures caused by new faults). In the case of a recurrence for which the underlying cause was identified, the diagnostic tool can rapidly identify a solution. In the case of a recurrence for which the underlying cause is being investigated, the diagnostic tool can prevent a repeated diagnosis by identifying previous failures caused by the same fault. These benefits are not free. Misdiagnosis is harmful, because a single misdiagnosis can result in multiple additional failures. (Such a danger exists in diagnoses by analysts, also.) You should implement the proposed approach in a pilot. You should make measurements to determine how well the approach works in real environments and to make design trade-offs.

8.6 Model Identification and Analysis of Models

The data analyses discussed in the previous sections reveal the software reliability behavior in real environments. Specifically, they identify the model structure and the range of parameter values. You can use this information to tune existing analytic or simulation models and to build new models. Then you can use the new models to predict various reliability characteristics in a new design by evaluating the model characteristics with a different set of parameters. This section discusses the modeling of the impact of software failures on performance, software reliability modeling in the operational phase, modeling of error detection and recovery, and modeling of software error bursts.

8.6.1 Impact of failures on performance

One of the key measures in evaluating gracefully degraded systems is the impact of failures on system performance or service capacity. In

fault-tolerant systems, it is also important to evaluate the effectiveness of various fault-tolerance techniques implemented to enhance software reliability. This subsection discusses the modeling of the impact of software failures on performance. It also evaluates the operating system fault tolerance achieved due to the built-in, single-failure tolerance in the Tandem system by conducting Markov reward analysis (App. B). Figure 8.12 shows a Markov model built using processor halt logs collected from a 16-processor Tandem system [Lee92]. In the figure, S_i represents the system state in which there are i failed processors because of software faults and $r_{i,j}$ represents the transition rate from S_i to S_j.

Two reward functions are defined in the analysis (Eqs. (8.1) and (8.2)). In these equations, r_i represents the reward rate for S_i. The first function (SFT) reflects the fault tolerance of the Tandem system. In this function, the first processor halt does not cause any degradation. For additional processor halts, the loss of service is proportional to the number of processors halted. The second function (NSFT) assumes no fault tolerance. The difference between the two functions allows evaluation of the improvement in service due to the built-in fault tolerance mechanisms.

SFT (single-failure tolerance):

$$r_i = \begin{cases} 1 & \text{if } i = 0 \\ 1 - \dfrac{i-1}{16} & \text{if } 0 < i < 16 \\ 0 & \text{if } i = 16 \end{cases} \qquad (8.1)$$

NSFT (no single-failure tolerance):

$$r_i = 1 - \frac{i}{16} \qquad \text{if } 0 \le i \le 16 \qquad (8.2)$$

Based on the above reward functions, the expected steady-state reward rate is evaluated for software, nonsoftware, and all halts. The

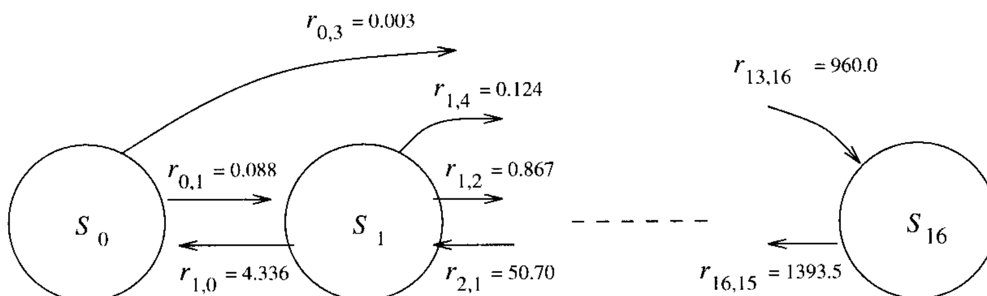


Figure 8.12 Measurement-based Markov model.

steady-state reward rate represents the relative amount of useful service the system can provide per unit of time in the long run; it is a measure of service-capacity-oriented software availability. The steady-state reward-loss rate (or simply, reward loss) represents the relative amount of useful service lost per unit of time due to processor halts.

The results of analysis are given in Table 8.11. Rows with "SFT" and "NSFT" show the estimated steady-state reward-loss with SFT and NSFT, respectively. The bottom row shows the improvement in service (i.e., reduction in reward loss) due to the fault tolerance. You can see that the single-failure tolerance in the measured system reduces the service loss due to software halts by 89 percent and the service loss due to nonsoftware halts by 92 percent. This clearly demonstrates the effectiveness of the implemented fault-tolerance mechanisms against software failures as well as nonsoftware failures. You can also see that software problems account for 30 percent of the service loss in the measured system (with SFT).

8.6.2 Reliability modeling in the operational phase

Software reliability models typically attempt to relate the history of fault identification during the development phase, verification efforts, and operational profile [Musa87, Rama82]. Usually, it is assumed that software is an independent entity and each identified fault has the same impact. However, our results indicate that there are other factors that significantly affect software reliability in real environments. First, a single, highly visible software fault can cause many failures, and recurrences can seriously degrade software reliability in the field. Second, for a class of software, the fault tolerance of the overall system can significantly improve software reliability by making the effects of software faults invisible to users. Clearly, reliability issues for operational software in general can be quite different from those for the software in the development phase.

TABLE 8.11 Loss of Service Caused by Processor Halts in the Tandem System

Measure		Software	Nonsoftware	All
NSFT	Reward	0.00062	0.00205	0.00267
	Loss	23.2	76.8	100
SFT	Reward	0.00007	0.00016	0.00023
	Loss	30.4	69.6	100
Improvement		89%	92%	91%

This subsection discusses the factors that determine software reliability in the operational phase, using a case study of the Tandem GUARDIAN operating system [Lee94b].

8.6.2.1 Model construction. A hypothetical eight-processor Tandem system whose software reliability characteristics are described by the parameters in Table 8.12 was considered. Here the term *software reliability* means the reliability of an overall system when only the faults in the system software are considered. A system failure was defined to occur when more than half of the processors in the system failed. All parameters in the table except λ and μ were estimated based on the measured data (Secs. 8.4.2 and 8.5.3). The values of λ and μ were determined to mimic the 30 years of software mean-time-between-failures (MTBF) and the mean-time-to-repair (MTTR) characteristics reported in [Gray90]. Thus, the objectives of the analysis are to model and evaluate reliability sensitivity to various factors, not to estimate the absolute software reliability.

With the use of process pairs, a fault in the Tandem system software can cause a single or double processor halt. Also, a double processor halt can cause additional processor halts if the two halted processors control key system resources that are needed by other processors. (Refer to [Lee95] for further details.) In Table 8.12, "P(double CPU halt | software failure)" is the probability that a double processor halt (i.e., the failure of a process pair) occurs given that a software failure occurs. A software failure refers to a processor halt due to software. Similarly, "P(system failure | double CPU halt)" is the probability that a system failure occurs given that a double processor halt occurs. These two parameters are used to describe the major failure mode of the system because of software. The parameter "P(system failure | single CPU halt)" represents the secondary failure mode, which captures single processor halts severe enough to cause system coldloads. The table shows these probabilities for first occurrences, recurrences, and unidentified failures. Unidentified failures refer to the cases in which

TABLE 8.12 Estimated Software Reliability Parameters

Failures	First occurrence	Recurrence	Unidentified
Failure rate	$\lambda_f = 0.24\lambda$	$\lambda_r = 0.61\lambda$	$\lambda_u = 0.15\lambda$
P(double CPU halt | software failure)	$C_{df} = 0.23$	$C_{dr} = 0.18$	$C_{du} = 0.0$
P(system failure | double CPU halt)	$C_{sdf} = 0.44$	$C_{sdr} = 0.63$	$C_{sdu} = 0.0$
P(system failure | single CPU halt)	$C_{ssf} = 0.05$	$C_{ssr} = 0.0$	$C_{ssu} = 0.0$

Failures:
Software failure rate = λ = 0.32/year

Recovery:
Recovery rate = μ = 1/hour

analysts believed that the underlying problems are software faults but had not yet located the faults.

Figure 8.13 shows the Markov model. In the model $S_i, i = 0, \ldots, 4$ represents that i processors are halted because of software faults. A system failure is represented by the S_{down} state. To evaluate software reliability, no recovery from a system failure is assumed. That is, the system failure state is an absorption state. The R_i state represents an intermediate state in which the system tries to recover from an additional software failure (ith processor halt) using process pairs.

If a software failure occurs during the normal system operation (i.e., when the system is in the S_0 state), the system enters the R_1 state. If the failure is severe enough to cause a system coldload, a system failure occurs; otherwise, the system attempts to recover from the failure by using backup processes located in other processors. If recovery is successful, the system enters the S_1 state; otherwise, a double processor halt occurs. If the two halted processors control key system resources (such as a set of disks) that are essential for system operation, the rest of the processors in the system also halt and a system failure occurs; otherwise, the system enters the S_2 state and continues to operate. The value of r, the transition rate out of an R_i, is small and has virtually no impact on software reliability; a value of one transition per minute is used in the analysis.

In Fig. 8.13, the three coverage parameters C_d, C_{sd}, and C_{ss} are calculated from Table 8.12:

$$C_d = \text{P(double CPU halt} \mid \text{software failure)} = \frac{\lambda_f C_{df} + \lambda_r C_{dr} + \lambda_u C_{du}}{\lambda_f + \lambda_r + \lambda_u} \tag{8.3}$$

$$C_{sd} = \text{P(system failure} \mid \text{double CPU halt)} = \frac{\lambda_f C_{df} C_{sdf} + \lambda_r C_{dr} C_{sdr} + \lambda_u C_{du} C_{sdu}}{\lambda_f C_{df} + \lambda_r C_{dr} + \lambda_u C_{du}} \tag{8.4}$$

and

$$C_{ss} = \text{P(system failure} \mid \text{single CPU halt)} = \frac{\lambda_f C_{ssf} + \lambda_r C_{ssr} + \lambda_u C_{ssu}}{\lambda_f + \lambda_r + \lambda_u} \tag{8.5}$$

From the model in Fig. 8.13, you can evaluate software reliability (i.e., the distribution of the time for the system to be absorbed to the system failure state, starting from the normal state). You can use tools such as SHARPE [Sahn87, Sahn95] to evaluate the distribution.

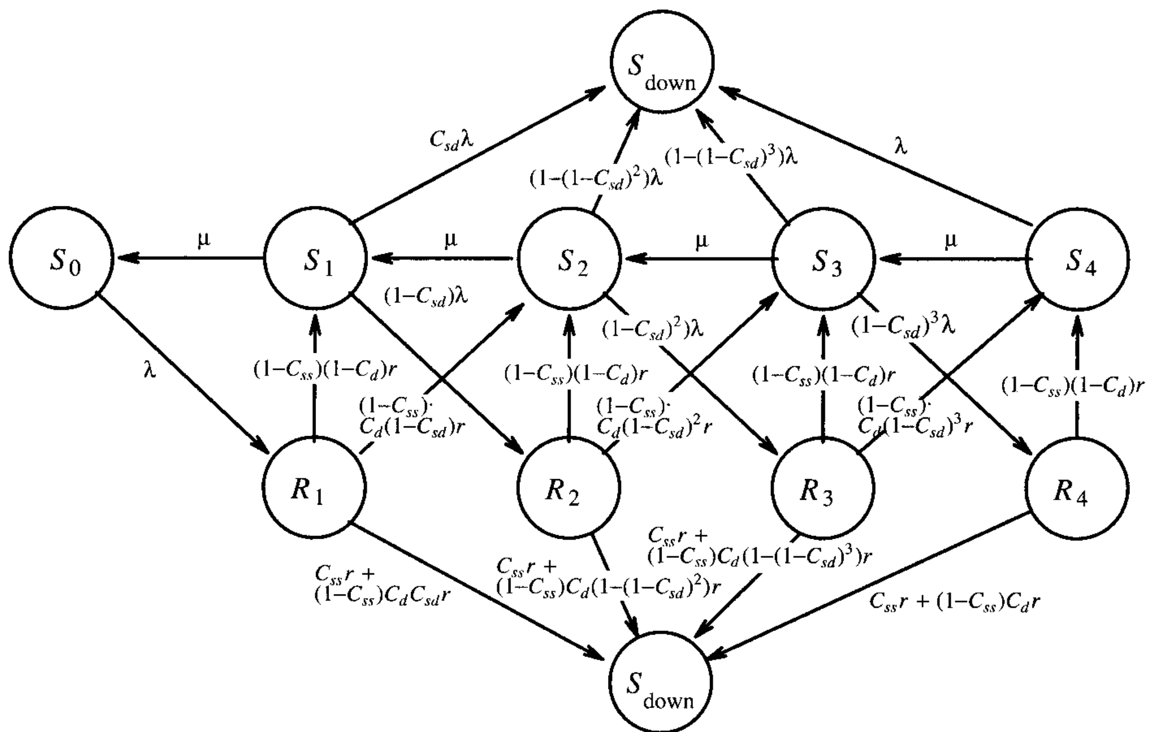


Figure 8.13 System-level software reliability model.

8.6.2.2 Sensitivity analysis. Table 8.13 shows the six factors considered. The second column of the table shows activities related to these factors, and the third column shows the model parameters affected by the factors. The coverage parameters C_d and C_{sd} are determined primarily by the robustness of process pairs and the system configuration, respectively. For example, C_d can be reduced by conducting extra testing of the routines related to job takeover. The parameter C_{sd} is primarily determined by the location of failed process pairs and the disk subsystem configuration. The recovery rate μ can be improved by automating the data collection and reintegration process.

Figure 8.14 shows the software MTBF evaluated using the model in Fig. 8.13 while varying the six factors in Table 8.13, one at a time. You can see that C_d and C_{sd} are almost as important as λ in determining the

TABLE 8.13 Factors of Software Reliability

Factor	Activity	Related parameters	
		Detailed	Overall
Software failure rate	Software development	λ_f, λ_r, λ_u	λ
Recurrence rate	Software service	λ_r	λ, C_d, C_{sd}, C_{ss}
Coverage parameter C_d	Robustness of process pairs	C_{df}, C_{dr}, C_{du}	C_d
Coverage parameter C_{sd}	System configuration	C_{sdf}, C_{sdr}, C_{sdu}	C_{sd}
Coverage parameter C_{ss}	—	C_{ssf}, C_{ssr}, C_{ssu}	C_{ss}
Recovery time	Diagnosability/maintainability	μ	μ

software MTBF. For example, a 20 percent reduction in C_d or C_{sd} has as much impact on software MTBF as an 18 percent reduction in λ. (The figure shows that the impact is approximately a 20 percent increase in software MTBF.) This result is understandable because the system fails primarily because of a double processor halt causing a set of disks to become inaccessible, not because of multiple independent software failures. You can also see that the recurrence rate has a significant impact on software reliability. A complete elimination of recurrences ($\lambda_r = 0$ in Table 8.12) would increase the software MTBF by a factor of 3.

Typically, it is assumed that the number of faults in software is the only major factor determining software reliability. Figure 8.14 clearly shows that in the Tandem system there are four degrees of freedom in improving software reliability: the number of faults in software, the recurrence rate, the robustness of the process pairs, and the system configuration strategy. The first two are general factors, and the last two are system-specific factors. Efforts to improve software reliability can be optimized by estimating the cost of improving each of the four factors.

8.6.3 Error/failure/recovery model

This subsection discusses the modeling of the detailed error detection and recovery processes in an operating system, using the data from the IBM MVS system running on an IBM 3081 mainframe [Hsue87]. The MVS system attempts to correct software errors using recovery routines. The philosophy in MVS is that for major system functions the designer envisions possible failure scenarios and writes a recovery routine for each. It is, however, the responsibility of the installation (or the user) to write recovery routines for applications.

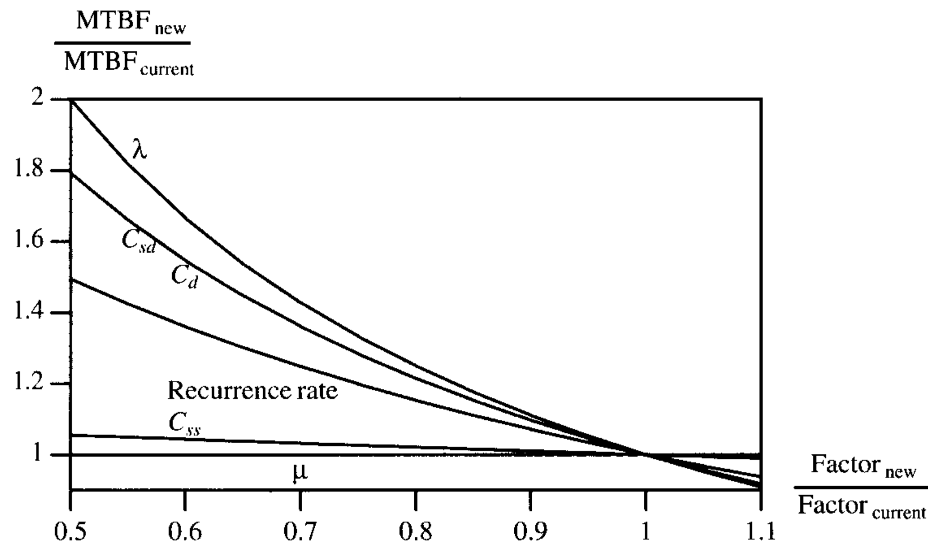


Figure 8.14 Software MTBF sensitivity.

More than one recovery routine can be specified for the same program. If the current recovery routine is unable to restore a valid state, the recovery manager can give control to another recovery routine, if available. This process is called *percolation.* The percolation process ends if a routine issues a valid retry request or if no more recovery routines are available. An error recovery can result in any of the following four situations:

1. Resume op (resume operation). The system successfully recovers from the error and returns control to the interrupted program.
2. Task term (task termination). The program and its related subtasks are terminated, but the system does not fail.
3. Job term (job termination). The job in control at the time of the error is aborted.
4. System failure. The job or task, which was terminated, is critical for system operation. As a result of the termination, a system failure occurs.

8.6.3.1 Model construction. The model consists of eight types of error states (Table 8.14) and three states resulting from error recoveries. Figure 8.15 shows the model, where a circle represents a state and an arrow represents a transition with an associated transition probability. The normal state represents the operating system running error-free. Note that the system failure state is not shown in the figure. This is because the occurrence of system failure was rare, and the number of observed system failures was statistically insignificant. Given that the system is in state i, the probability that it will go to state j, p_{ij}, can be estimated from the data as follows:

$$p_{ij} = \frac{\text{observed number of transitions from } E_i \text{ to } E_j}{\text{observed number of transitions out of } E_i} \tag{8.6}$$

TABLE 8.14 Mean Waiting Time

State	No. observations	Mean waiting time (sec.)	Standard deviation
Normal (error-free)	2757	10461.33	32735.04
CTRL (control error)	213	21.92	84.21
DLCK (deadlock)	23	4.72	22.61
I/O (I/O & data management error)	1448	25.05	77.62
PE (program exception)	65	42.23	92.98
SE (storage or address exception)	149	36.82	79.59
SM (storage management error)	313	33.40	95.01
OTHR (other type)	66	1.86	12.98
MULT (multiple software error)	481	175.59	252.79

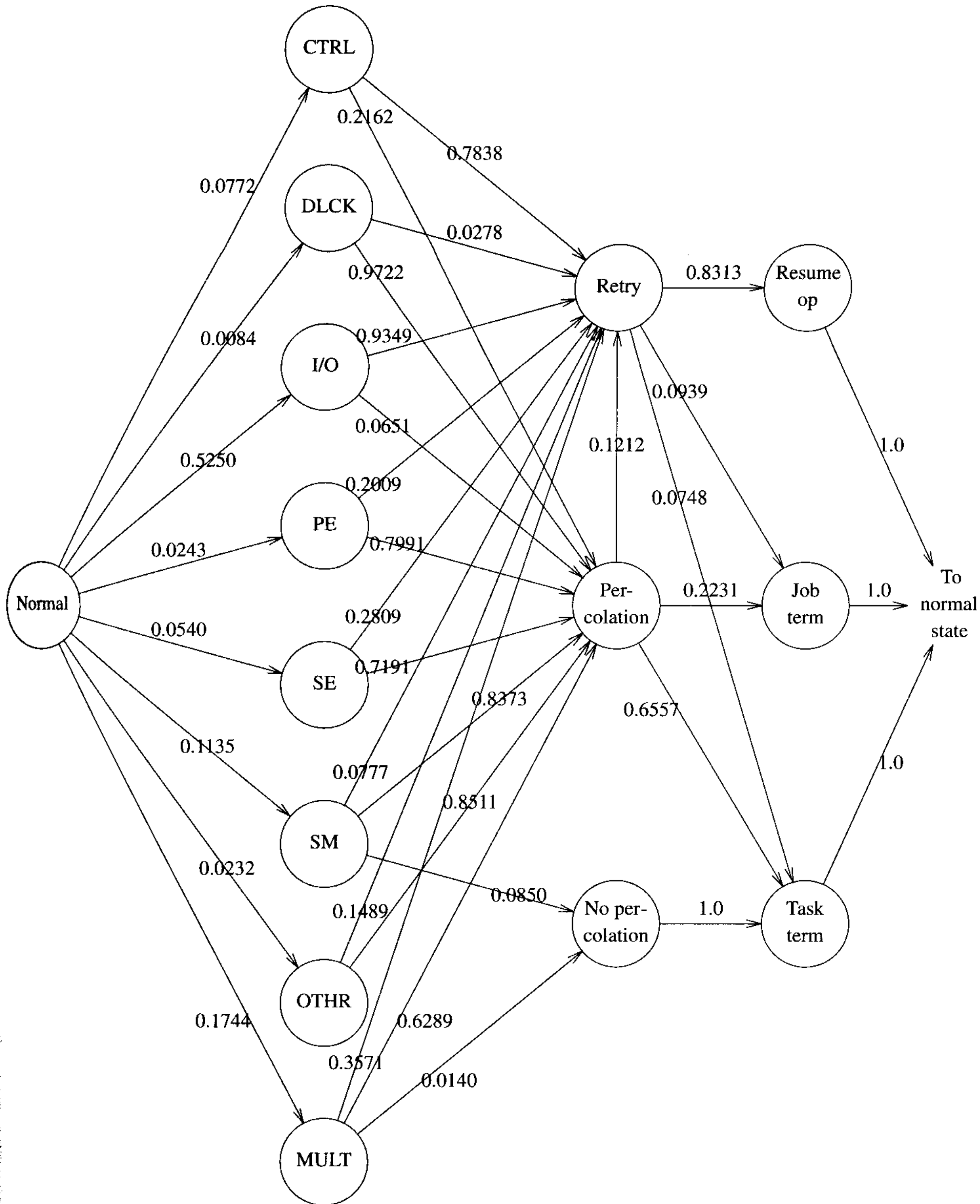


Figure 8.15 MVS software error/recovery model.

An error recovery can be as simple as a retry or as complex as requiring several percolations before a successful retry. The problem can also be such that no retry or percolation is possible. You can see from Fig. 8.15 that about 83.1 percent of all retries are successful. You can also see that 93.5 percent of I/O and data management errors and 78.4 per-

cent of control-related errors resulted in a direct retry. These observations indicate that most I/O- and control-related errors are relatively easy to recover from compared to the other types of errors, such as deadlock or storage errors.

Table 8.14 shows the mean-waiting-time characteristics of the normal and error states in the model. (See Sec. 8.4.2 for the definitions of error types.) You can see that the average duration of a multiple error is at least four times longer than that of any type of single error, which is typically in the range of 20 to 40 seconds, except for DLCK (deadlock) and OTHR (others). The average recovery time from a program exception is twice as long as that from a control error (21 seconds versus 42 seconds). This is probably due to the extensive software involvement in recovering from program exceptions.

8.6.3.2 Model evaluation. Table 8.15 shows the following steady-state measures evaluated from the model. The detailed definitions of these measures are given in [Howa71].

Transition probability (π_j)	Probability that the transition is to state j, given a transition to occur
Occupancy probability (Φ_j)	Probability that the system occupies state j at any time point
Mean recurrence time ($\overline{\Theta}_j$)	Mean recurrence time of state j

The occupancy probability of the normal state can be viewed as the operating system availability without degradation. The state transition probability, on the other hand, characterizes error-detection and recovery processes in the operating system. Table 8.15*a* lists the state transition probabilities and occupancy probabilities for the normal and error states. Table 8.15*b* lists the state transition probabilities and the mean recurrent times of the recovery and result states. A dash (—) in the table indicates a negligible value (less than 0.00001). You can see that the occupancy probability of the normal state in the model is 0.995. This indicates that 99.5 percent of the time the operating system is running error-free. In the other 0.5 percent of the time, the operating system is in an error or recovery state. In more than half of the error and recovery time (that is, 0.29 percent out of 0.5 percent) the operating system is in the multiple-error state.

By solving the model, you can find that the operating system makes a transition every 43.37 minutes. Table 8.15*a* shows that 24.74 percent of all transitions made in the model are to the normal state, 24.73 percent to error states (obtained by summing the π's for all error states), 25.79 percent to recovery states, and 24.74 percent to result states. Since a transition occurs every 43 minutes, you can estimate that, on

TABLE 8.15 Error/Recovery Model Characteristics

Measure	Normal state	Error state							
		CTRL	DLCK	I/O	PE	SE	SM	OTHR	MULT
π	0.2474	0.0191	0.0020	0.1299	0.0060	0.0134	0.0281	0.0057	0.0431
Φ	0.9950	0.00016	—	0.00125	0.000098	0.000189	0.00036	—	0.002913

(a)

Measure	Recovery state			Result state		
	Retry	Percolation	No-percolation	Resume op	Task term	Job term
π	0.1704	0.0845	0.0030	0.1414	0.0712	0.0348
$\overline{\Theta}$(hr.)	4.25	8.55	241.43	5.11	10.16	20.74

(b)

the average, a software error is detected every 3 hours and a successful recovery (i.e., reaching the "resume op" state) occurs every 5 hours. Table 8.15*b* also shows that more than 40 percent of software errors lead to job or task terminations, thus causing the loss of service to users. However, only a few of these terminations lead to system failures. This result indicates that recovery routines in MVS are effective in avoiding system failures, but are not so effective in avoiding user job terminations.

8.6.4 Multiple-error model

The error/failure/recovery model and analysis using the model in Sec. 8.6.3 showed that multiple errors are a significant source of system degradation in the MVS system. Figure 8.16 shows a semi-Markov model for a multiple error developed from the data from an IBM MVS system [Hsue87] (Secs. 8.4.2 and 8.4.4). The model was constructed assuming zero waiting time in the normal state (i.e., assuming the occurrence of a multiple error). The figure not only illustrates the interactions among different software errors, but also provides detailed information on the occurrence of transitions. For example, if a program exception error (PE) occurs, there is about a 63 percent chance that a storage exception error (SE) will follow. Further, there is about a 50 percent chance that a storage error (SE or SM) will be followed by another error of the same type.

Table 8.16 lists the characteristics for a multiple error obtained by solving the semi-Markov model described in Fig. 8.16 with a zero holding time in the normal state (i.e., given that a multiple error occurs). In the table, e_j (entry probability) represents the probability that the system enters state j, given an entrance to occur [Howa71]. You can see (from π, transition probability) that nearly 30 percent of the transitions are made to the storage exception state when the system enters a multiple error mode. Once in a multiple error mode, a storage exception error occurs every 1 minute and 45 seconds ($\overline{\Theta} = 0.0292$ hours in Table 8.16), while the average duration of multiple errors is about 2 minutes and 56 seconds ($\overline{\Theta} = 0.0489$ hours, the recurrence time of the normal state). Note that the average duration of a multiple error predicted here from the model is very close to the mean duration of a multiple error, 175.5 seconds obtained from the real data, listed in Table 8.14. This provides evidence that the semi-Markov process is a good model for the measured system. As soon as an entry into a multiple error is made, consecutive errors are detected almost every 31 seconds (by taking the reciprocal of the sum of all entry probabilities e in Table 8.16). This indicates that about five to six errors will be detected, on average, once a multiple error occurs.

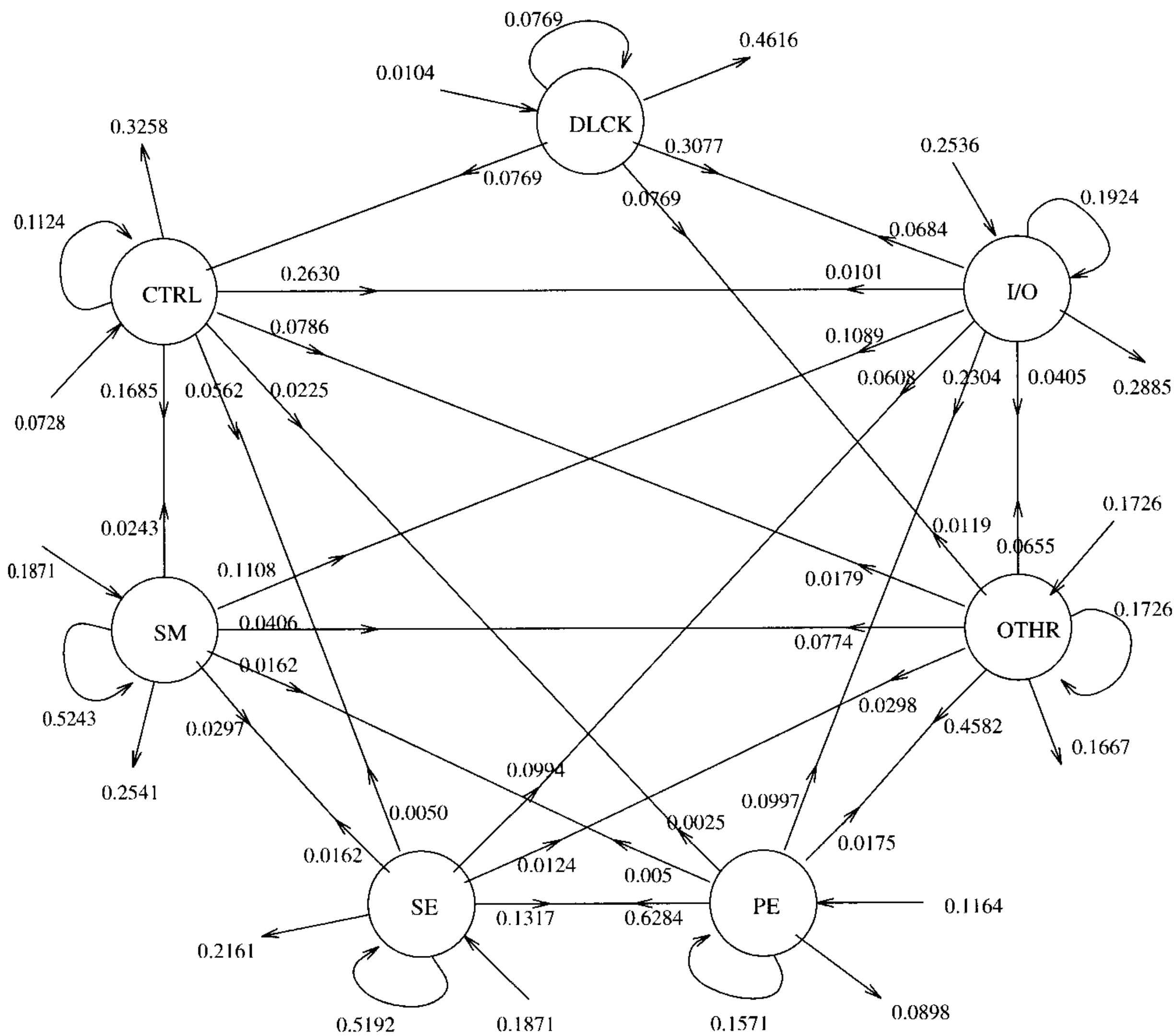


Figure 8.16 State transition diagram of multiple error (MULT).

8.7 Impact of System Activity

This section discusses the relationships between software failures and various workload parameters. Several studies have shown that you cannot consider software reliability in real environments without taking the system workload into account. [Cast81, Cast82,] and [Iyer82a] proposed analytic or regression models of such relationships. Markov models of such relationships have been proposed in [Hsue88].

8.7.1 Statistical models from measurements

8.7.1.1 Workload-dependent cyclostationary model. An early study [Cast81] introduced a workload-dependent cyclostationary model to characterize system failure processes. The model is based on the observation of the periodic nature of daily workload profile and failures. The

TABLE 8.16 Characteristics of Multiple Errors

Measure	Normal state	Error state							
		CTRL	DLCK	I/O	PE	SE	SM	OTHR
π	0.1767	0.0327	0.0048	0.1451	0.1473	0.2957	0.1360	0.0617
Φ	0	0.0648	0.0130	0.3004	0.0837	0.2202	0.2717	0.0462
$\underline{e}$	0.00568	0.00105	0.00015	0.00466	0.00473	0.00950	0.00437	0.00198
Θ (hr)	0.0489	0.2647	1.8126	0.0596	0.0587	0.0292	0.0636	0.1401

underlying idea is that a higher workload implies that the kernel of the operating system is exercised more per unit of time, increasing the probability of system failure. It is assumed that the instantaneous failure rate of a system resource can be approximated by a function of the usage of the resource considered. Specifically, the failure rate of a particular resource, $\lambda(t)$, is assumed to be

$$\lambda(t) = au(t) + b \tag{8.7}$$

where $u(t)$ is a usage function of the resource that, in turn, consists of a deterministic, periodic function of time, $m(t)$ and a modified, stationary gaussian process, $z(t)$:

$$u(t) = m(t) + z(t) \tag{8.8}$$

The failure arrivals are assumed to follow a Poisson process. Thus, the failure process involves two stochastic processes: a Poisson process and a gaussian process. Such a process is defined as a *doubly stochastic process.* The following workload-dependent cyclostationary reliability function due to software is derived:

$$R(t) = 1 - \phi(t)e^{-k_1 t - k_2(1 - e^{-k_3 t}) - k_4(1 - e^{-k_5 t})} \tag{8.9}$$

where $\phi(t)$ is a periodic function of time, depending on the periodic component of $\lambda(t)$ and k_1, k_2, k_3, k_4, and k_5 are constants determined from the failure and usage process characteristics (Eqs. (8.7) and (8.8)).

8.7.1.2 Load hazard model. In [Iyer82a], a load hazard model was introduced to measure the risk of a failure as the system activity increases. The proposed model is similar to the hazard rate defined in reliability theory. Given a workload variable X, the load hazard is defined as

$$z(x) = \frac{P[\text{failure in load interval } (x, x + \Delta x)]}{P[\text{no failure in load interval } (0, x)]\ \Delta x} = \frac{g(x)}{1 - G(x)} \tag{8.10}$$

where $g(x)$ is the probability density function (pdf) of the variable "a failure occurs at a given workload value x" and $G(x)$ is the corresponding cumulative distribution function (cdf). That is,

$$g(x) = P(\text{failure occurs} \mid X = x) = \frac{f(x)}{l(x)} \tag{8.11}$$

where $l(x)$ is simply the pdf of the workload in consideration:

$$l(x) = P(X = x) \tag{8.12}$$

and $f(x)$ is the joint pdf of the system failure and the workload:

$$f(x) = P(\text{failure occurs and } X = x) \tag{8.13}$$

The load hazard $z(x)$ (in close analogy with the classical hazard rate in reliability theory) measures the incremental risk involved in increasing the workload from x to $x + \Delta x$. A constant hazard rate implies that failures are occurring randomly with respect to the workload. An increasing hazard rate on the increase of X implies that there is an increasing failure rate with increasing workload.

The load hazard model was applied to the software failure and workload data collected from an IBM 3081 system running the VM operating system [Iyer85b]. Based on the collected data, $l(x)$, $f(x)$, $g(x)$, and $z(x)$ were computed for each workload variable. Figure 8.17 shows the $z(x)$ plots for three selected workload variables:

OVERHEAD	The fraction of CPU time spent on the operating system
PAGEIN	The number of page reads per second by all users
SIO (start I/O)	The number of input/output operations per second

The regression coefficient, R^2, which is an effective measure of the goodness of fit, is also shown in the figure.

You can see from the hazard plots shown that the workload parameters appear to be acting as stress factors, i.e., the failure rate increases as the workload increases. The effect is particularly strong in the case

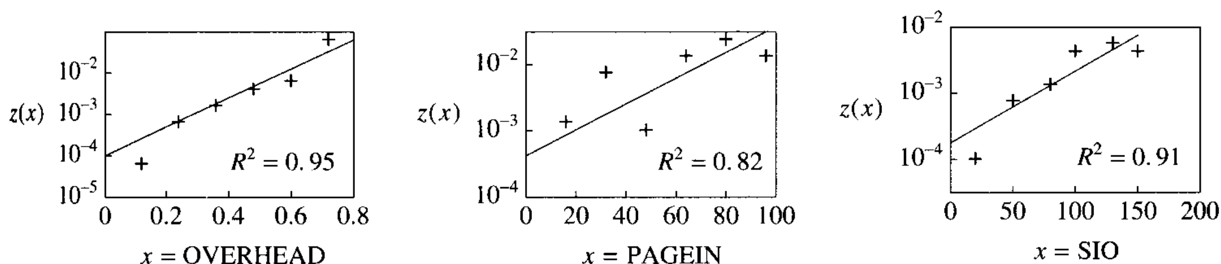


Figure 8.17 Workload hazard plots for an IBM 3081 system.

of the interactive workload measures OVERHEAD and SIO. The correlation coefficients of 0.95 and 0.91 show that the failure process closely fits an increasing load hazard model. The risk of a failure also increases with increased PAGEIN, although at a somewhat lower correlation (0.82). Note that the vertical scale on these plots is logarithmic, indicating that the relationship between the load hazard $z(x)$ and the workload variable is exponential, i.e., the risk of a software failure increases exponentially with increasing workload.

It was hypothesized that, in addition to the reasons reported in [Cast81], there are other load-induced effects [Iyer82b]. The first is the latent discovery effect. Problems must be detected in order to cause failures. Even if failures may not be caused by increased workload, they are revealed by this factor. The second effect is the load-induced software failures. Many typical software faults exist in the operating system. These faults can be divided in two groups, those triggered under high load and those that are load-independent but appear to be load-induced because of an increased execution time effect.

8.7.2 Overall system behavior model

This subsection introduces a measurement-based performability model based on error and resource-usage data collected on a production IBM 3081 system running under the MVS operating system [Hsue88].

8.7.2.1 Workload model. The workload data were collected by sampling, at predetermined intervals, four resource usage meters:

CPU	The fraction of the measured interval for which the CPU is executing instructions
CHB	The fraction of the measured interval for which the channel is busy and the CPU is in the wait state (commonly used to measure the degree of contention in a system)
SIO	The number of successful start I/O and resume I/O instructions issued to the channel
DASD	The number of requests serviced on the direct-access storage device

At any interval of time, the measured workload is represented by a point in a four-dimensional space (CPU, CHB, SIO, DASD). Statistical cluster analysis (App. B) is used to divide the workload into similar classes according to a predefined criterion. This allows you to concisely describe the dynamics of system behavior and extract a structure that already exists in the workload data. Each cluster (defined by its centroid) is then used to depict a system state, and a state-transition dia-

gram (consisting of intercluster transition probabilities and cluster sojourn times) is developed. A k-means algorithm [Spat80] is used for clustering.

Figure 8.18 shows the workload model built by the above procedures using the CPU and CHB data. This combination was found to best describe the CPU-bound load, while models based on SIO and DASD were found to best describe the I/O workload. Note that the null state W_0 has been incorporated to represent the state of the system during the nonmeasured period. (The measurements were not made continuously during the entire measurement period.) The time spent in the null state is assumed to be zero. Table 8.17 shows the results of the clustering operation. You can see that for about 36 percent of the time the CPU was heavily loaded (0.96), and for 76 percent of the time the CPU load was above 0.5.

8.7.2.2 Resource-usage/error/failure model. Error data during the measurement period were passed through a coalescing algorithm and then through an additional reduction technique based on the probabilistic relationships between errors [Iyer90]. The resulting errors were then classified into five classes:

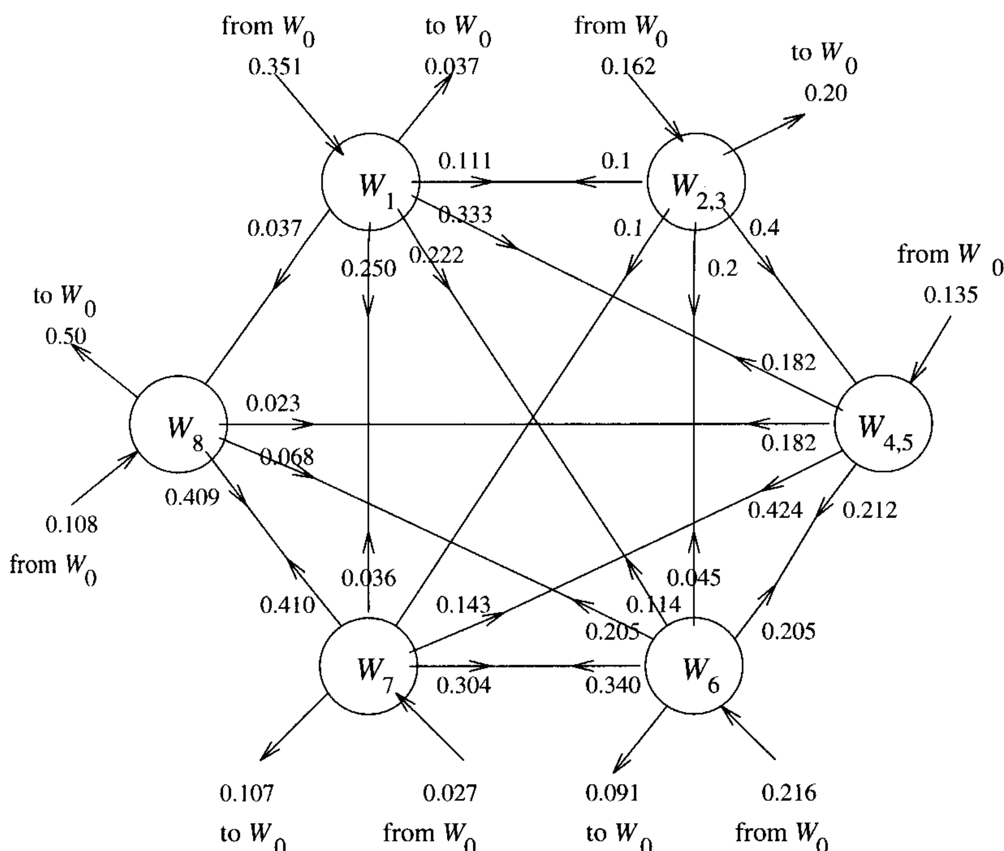


Figure 8.18 State transition diagram of CPU-bound load.

TABLE 8.17 Characteristics of CPU-Bound Workload Clusters

Cluster id	% of obs	Mean of CPU	Mean of CHB	Std. dev. of CPU	Std. dev. of CHB
W_1	7.44	0.0981	0.1072	0.0462	0.0436
W_2	0.50	0.1126	0.5525	0.0433	0.0669
W_3	2.73	0.1547	0.2801	0.0647	0.0755
W_4	12.41	0.3105	0.1637	0.0550	0.0459
W_5	0.74	0.3639	0.3819	0.0365	0.1923
W_6	17.12	0.5416	0.1287	0.0560	0.0511
W_7	22.58	0.7207	0.0848	0.0576	0.0301
W_8	36.48	0.9612	0.0168	0.0362	0.0143

R^2 of CPU = 0.9724
R^2 of CHB = 0.8095
overall R^2 = 0.9604
(R^2: the square of correlation coefficient)

1. CPU — Errors that affect the normal operation of the CPU; may originate in the CPU, in the main memory, or in a channel
2. CHAN — Channel errors (the great majority are recovered)
3. DASD — Disk errors, recoverable (by data correction or instruction retry) and nonrecoverable
4. SWE — Software incidents due to invalid supervisor calls, program checks, and other exceptions
5. MULT — Multiple errors that affect more than one type of component (i.e., involving more than one of the above)

The recovery procedures were divided into four categories based on recovery cost, which was measured in terms of the system overhead needed to handle an error. The lowest level (hardware recovery or HWR) involves the use of an error correction code (ECC) or hardware instruction retry; it has minimal overhead. If hardware recovery is not possible or unsuccessful, software-controlled recovery (SWR) is invoked. This could be simple (e.g., terminating the current program or task in control) or complex (e.g., invoking a specially designed recovery routine(s) to handle the problem). The third level, alternative (ALT), involves transferring the tasks to functioning processor(s) when one of the processors experiences an unrecoverable error. If no on-line recovery is possible, the system is brought down for off-line (OFFL) repair.

The separate workload, error, and recovery models developed were combined into a single model, shown in Fig. 8.19. Due to the complexity of the entire model, the figure shows only a part of the model. The null state W_0 is not shown in the diagram. The model in Fig. 8.19 captures the workload-dependent error and recovery process in the system. The model has three classes of state: normal operation states (S_N),

error states (S_E), and recovery states (S_R). Under normal conditions, the system makes transitions from one workload state to another. The occurrence of an error results in a transition to one of the error states. The system then goes into one or more recovery states after which, with a high probability, it returns to one of the workload states. You can see from the state transition diagram that nearly 98.3 percent of hardware recovery requests and 99.7 percent of software recovery requests are successful.

8.7.2.3 Performability analysis. The resource-usage/error/recovery model was used to evaluate the performability of the system. Reward functions were used to depict the performance degradation due to errors and due to different types of recovery procedure (App. B). Since the recovery overhead for each error event in the modeled system is approximately constant, the total recovery overhead, and thus the reward, depends on the error rate during the event. On this basis, the reward rate r_i (per unit time) for each state of the model is defined as

$$r_i = \begin{cases} \dfrac{s_i}{s_i + e_i} & \text{if } i \in S_N \cup S_E \\ 0 & \text{if } i \in S_R \end{cases} \qquad (8.14)$$

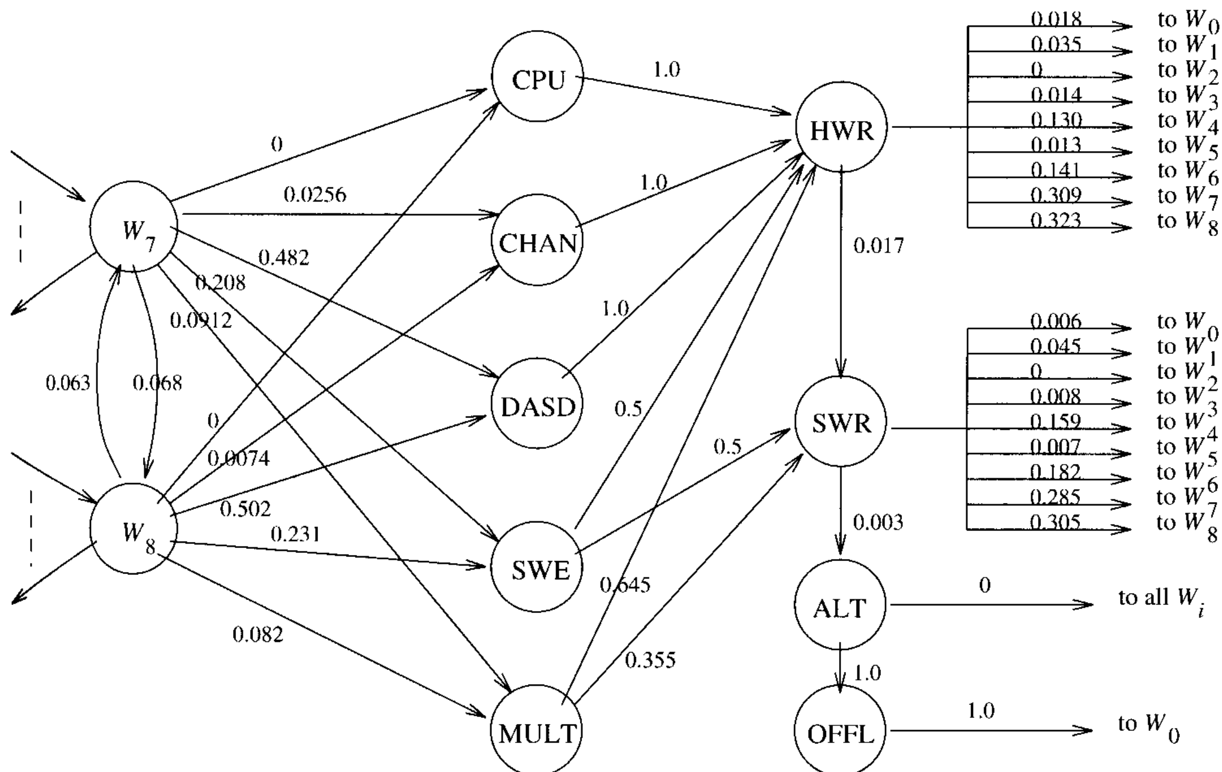


Figure 8.19 State transition diagram of resource-usage/error/recovery model.

where s_i and e_i are the service rate and the error rate in state i, respectively. Thus one unit of reward is given for each unit of time when the system stays in the normal states S_N. The reward rate decreases with increasing number of errors generated in an error state. Zero reward is assigned to recovery states.

The reward rate of the modeled system at time t is a random variable $X(t)$. Therefore, the expected reward rate $E[X(t)]$ can be evaluated from $E[X(t)] = \Sigma_i p_i(t) r_i$ where $p_i(t)$ is the probability of being in state i at time t. The cumulative reward by time t is $Y(t) = \int_0^t X(\sigma)d\sigma$, and the expected cumulative reward is given by

$$E[Y(t)] = E\left(\int_0^t X(\sigma)d\sigma\right) = \sum_i r_i \int_0^t p_i(\sigma)d\sigma \qquad (8.15)$$

The impact of different types of errors were evaluated by calculating the expected reward rate with different definitions of absorption state. In Fig. 8.20, "OFFL" represents the expected reward rate when only off-line repairs are considered as an absorption state. This curve actually represents the system reliability. "MULT" represents the expected reward rate when off-line repairs and multiple errors are considered as an absorption state. The difference between the two curves captures the impact of multiple errors on performability.

8.8 Summary

In this chapter, we discussed the current issues in the area of measurement-based analysis of software reliability in the operational phase. The discussion centered around techniques, our experiences, and major developments in this area. This chapter addressed measurement techniques, analysis of data, model identification, analysis of models, and the effects of workload on software reliability. For each field, we discussed the key issues and then presented detailed techniques and representative work.

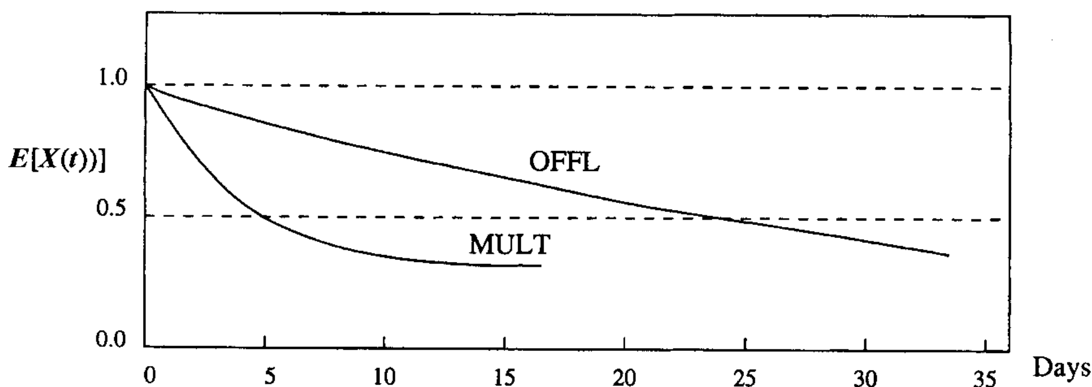


Figure 8.20 Expected reward rate.

Significant progress has been made in all these fields over the past 15 years. Increasing attention is being paid to: (1) combining analytical modeling and experimental analysis and (2) combining software design and evaluation. In the first aspect, state-of-the-art analytical modeling techniques are being applied to real operational software to evaluate various reliability and performance characteristics. Results from experimental analysis are being used to validate analytical models and to reveal practical issues that analytical modeling must address to develop more representative models. In the second aspect, software failure data from the operational phase are being used to identify and address software design issues for improving software reliability. Efficient on-line recovery and off-line diagnosis techniques are being developed based on data collected from the operational phase. Further interesting studies and advances in this area can be expected in the near future.

Problems

8.1 Discuss the power and limitations of measurement-based approach, as compared with analytic or simulation-based approach.

8.2 Reliability characteristics of operational software can be quite different from those of the software in its development phase. List the factors that contribute to such differences.

8.3 Describe general steps to build a software reliability model from measurements. What is the use of a measurement-based model?

8.4 It has been shown that software errors tend to occur in bursts. Discuss the reasons for error bursts. What are the effects of error bursts on software reliability?

8.5 Software and hardware components of a computer system may not fail independently of each other. Discuss the reasons for such correlated failures. Discuss the techniques for analyzing two-way and multiway failure dependencies.

8.6 Checkpointing and restart implemented in distributed transaction processing environments can allow a system to tolerate certain software faults. What are the reasons for the software fault tolerance? What are the factors that determine the level of software fault tolerance achieved with the technique in such environments? What are the advantages and disadvantages of using the above technique as compared with using the techniques such as N-version programming and recovery blocks?

8.7 It has been shown that, in environments where many users run the same software, the majority of field software failures are recurrences. What are the

reasons for recurrences? What are the effects of recurrences on software reliability? How can we reduce such effects?

8.8 What are the effects of increased system activity on software reliability in an operational system? Discuss the reasons for such relationships.

8.9 Table 8.18 shows software time-to-failure (TTF) data measured on a multicomputer system. Table 8.19 shows the completion time (CT) of a benchmark running on a multicomputer system under different workload conditions. The time in Tables 8.18 and 8.19 represents the time at the end of the corresponding time interval. For example, the first column of Table 8.18 indicates that there are 29 TTF instances whose values are smaller than 1 day. Do the following work using the data:

a. Construct an empirical distribution for each set of data.

b. Fit both empirical distributions to an exponential function, the TTF distribution to a two-phase hyperexponential function, and the CT distribution to a two-phase hypoexponential function, respectively.

c. Test the goodness of fits using the chi-square test.

8.10 Table 8.20 (shown below and in the Data Disk) gives processor failure data collected from a distributed system consisting of five processors connected by a local network. Each record in the table has the following format:

Processor id	Failure time	Recovery time	Error type

The time unit is second and time 0 is 12 A.M., 10/1/1987. The measurement started from 12 A.M., 12/9/1987 (29,548,800), and ended at 12 A.M., 8/15/1988 (51,148,800), covering 250 days. The error type means the type of error that causes a processor failure. Possible error types are CPU, I/O (network or disk problems), software, and unknown. Do the following work using the data:

a. Obtain failure rate and recovery rate for each processor.

TABLE 8.18 Software TTF Data from a Multicomputer

TTF (days)	1	2	3	4	5	6	7	8	9	10	11	12	13
Frequency	29	8	6	5	3	3	1	3	1	1	1	2	1
TTF (days)	14	15	15	17	18	19	20	21	22	23	24	25	26
Frequency	0	2	0	1	0	0	0	0	0	0	0	0	1

TABLE 8.19 Completion Time of a Benchmark

Time (min.)	2	4	6	8	10	12	14	16	18	20	22	24
Frequency	2	10	12	18	16	13	15	7	6	4	2	6
Time (min.)	26	28	30	32	34	36	38	40	42	44	46	48
Frequency	4	1	0	1	4	0	0	1	1	0	0	0

TABLE 8.20 Processor Failure Data from a Five-Processor Multicomputer System

Processor id	Start time	End time	Error type
1	29604556	29605120	Software
1	29704893	29706486	I/O
1	29770774	29772334	Software
1	29779466	29779946	I/O
1	29918361	29919001	I/O
1	29938155	29938995	Unknown
1	29968514	29969314	I/O
1	30204850	30206947	I/O
1	30300136	30300482	I/O
1	30315829	30316496	I/O
1	30552159	30555408	Unknown
1	30571134	30571529	I/O
1	31762359	31762897	I/O
1	31830453	31831347	I/O
1	31837833	31838304	I/O
1	31839160	31839628	I/O
1	31951476	31952241	I/O
1	32123531	32124925	I/O
1	32126834	32127160	I/O
1	32177392	32178646	I/O
1	32963455	32964054	Software
1	33014870	33015438	I/O
1	33152933	33153737	Software
1	34616577	34617326	Software
1	34770846	34771459	I/O
1	37813703	37814561	Unknown
1	38007128	38049817	I/O
1	38955976	38956597	Unknown
1	38961843	38962658	Unknown
1	39465650	39467247	I/O
1	39809295	39810575	Unknown
1	40069978	40071607	I/O
1	41000249	41001273	Software
1	41366807	41387784	Software
1	41391480	41392113	Software
1	42272616	42273174	Software
1	42831896	42833058	Software
1	43309767	43313204	Software
1	43348292	43350322	Software
1	43952410	43953022	Software
1	44877091	44877998	Software
1	45841909	45842888	Software
1	46961851	46962724	Software
1	48878979	48880349	Unknown
1	48888392	48890586	Software
2	29570604	29570904	I/O
2	29577262	29577562	I/O
2	29767256	29767556	I/O
2	29782058	29782358	I/O
2	29788920	29789718	I/O
2	29886930	29887230	I/O

TABLE 8.20 ***(Continued)*** **Processor Failure Data from a Five-Processor Multicomputer System**

Processor id	Start time	End time	Error type
2	29909506	29909806	I/O
2	29910884	29911926	I/O
2	29913095	29913465	I/O
2	29914121	29915273	I/O
2	29916032	29916705	I/O
2	29917194	29917844	I/O
2	29918236	29919428	I/O
2	29939825	29940125	Unknown
2	29946828	29947128	Unknown
2	29949602	29950199	I/O
2	29953804	29954104	I/O
2	29957176	29957476	I/O
2	29963079	29963379	I/O
2	29964579	29966271	I/O
2	29967435	29968115	I/O
2	30550260	30550560	Unknown
2	30550946	30553816	Software
2	30660369	30660669	Unknown
2	31774557	31774857	I/O
2	31775859	31776450	I/O
2	31781002	31781783	I/O
2	31832367	31832781	I/O
2	31839101	31841182	I/O
2	32122929	32129036	I/O
2	32176731	32177250	Software
2	32178134	32180594	I/O
2	32183767	32184321	I/O
2	32449592	32450399	Unknown
2	32962291	32962591	Unknown
2	32975370	32975670	I/O
2	32976674	32976974	I/O
2	32983427	32983727	I/O
2	33049654	33050104	I/O
2	33052795	33053095	I/O
2	33057253	33057553	I/O
2	33059795	33060095	I/O
2	33087233	33087533	I/O
2	33089396	33089696	I/O
2	33120965	33121265	Unknown
2	33148035	33148430	I/O
2	34011900	34012200	Unknown
2	34770845	34771810	I/O
2	34774453	34774753	Unknown
2	34775145	34776270	I/O
2	34777013	34777313	I/O
2	34777969	34778269	I/O
2	34780806	34781106	I/O
2	35659389	35662510	Unknown
2	35668585	35668885	I/O
2	35725413	35725713	I/O
2	35726840	35727669	I/O

TABLE 8.20 ***(Continued)*** **Processor Failure Data from a Five-Processor Multicomputer System**

Processor id	Start time	End time	Error type
2	35730910	35731210	I/O
2	35744817	35745117	I/O
2	35753244	35753544	I/O
2	36785044	36785772	Unknown
2	36789532	36789988	Software
2	36796257	36847121	I/O
2	38249305	38249785	Software
2	38251655	38252099	Unknown
2	38744311	38744777	Software
2	38745674	38746341	CPU
2	38955454	38956597	Unknown
2	39805355	39808038	I/O
2	43064089	43065444	Unknown
2	44732995	44733453	Software
2	45221452	45221917	I/O
2	46375146	46375583	Unknown
2	49391794	49392273	Software
2	50068269	50069049	I/O
3	30550976	30554633	Software
3	31760114	31760624	Software
3	32806356	32806844	Software
3	33152933	33153558	Software
3	34370843	34385770	I/O
3	34779280	34779754	Unknown
3	36783938	36786522	I/O
3	36787771	36788262	Software
3	37800626	37811789	I/O
3	37812929	37813548	Software
3	43175442	43175990	Software
3	43326842	43327797	Unknown
3	43330318	43330785	CPU
3	43331708	43332312	CPU
3	43334898	43338477	CPU
3	43338720	43340663	I/O
3	43342983	43345110	CPU
3	43691509	43692018	Software
3	43693033	43693689	Unknown
3	43693580	43694622	I/O
3	43695584	43696455	Unknown
3	43697283	43698493	I/O
3	47651456	47651969	Software
3	49057997	49058626	Software
3	49568366	49568810	Unknown
3	50043031	50043500	Software
3	50697666	50698119	Unknown
3	50930312	50930764	Unknown
4	29939050	29940425	I/O
4	29943202	29948396	I/O
4	30550948	30554287	I/O
4	30762674	30762976	I/O
4	34777878	34778529	I/O

TABLE 8.20 ***(Continued)*** **Processor Failure Data from a Five-Processor Multicomputer System**

Processor id	Start time	End time	Error type
4	37563700	37567137	I/O
4	37811277	37811786	I/O
4	38956002	38963057	I/O
4	39204773	39205074	I/O
4	43175015	43175315	Unknown
4	47239532	47243366	I/O
4	47641698	47642345	I/O
4	48939418	48940072	Unknown
4	49144906	49145430	I/O
4	50662627	50663276	Unknown
4	50668168	50672456	I/O
4	50758111	50758751	I/O
4	50941271	50943381	I/O
5	30545108	30545606	Software
5	30551262	30553813	Unknown
5	31769662	31770215	Software
5	31953896	31954395	Unknown
5	33152335	33153779	Unknown
5	33153616	33156417	Software
5	34774852	34780711	I/O
5	37813181	37813664	Unknown
5	38955806	38962854	Unknown
5	40928410	40934268	Software
5	40936013	40936767	Unknown
5	43189066	43193507	Software
5	43252583	43254201	Unknown
5	43256847	43257890	Unknown
5	43328379	43333113	Software
5	44903066	44903945	Unknown
5	47064423	47069097	I/O
5	49125607	49126676	Software
5	49478273	49478888	Unknown
5	50618397	50623146	Software

b. Assuming failures on different processors are independent, build a Markov model based on the failure and recovery rates obtained in item *a*.

c. Assuming the modeled system is a three-out-of-five system, solve the model (using SHARPE [Sahn87, Sahn95] or similar tools) to obtain the reliability of the system. (Note: To obtain SHARPE, contact Professor Kishor Trivedi at Duke University. Phone: (919) 660-5269, e-mail: kst@ee.egr.duke.edu.)

d. Build a measurement-based Markov model using the data without assuming failures on different processors are independent.

e. Solve the measurement-based model (using SHARPE or similar tools) to obtain the reliability of the three-out-of-five system, and compare the result with that obtained in item *c*.

f. Construct a failure data matrix and then use the matrix to calculate correlation coefficient for each pair.