PCS - 5733

Design and Analysis of Fault-Tolerant Digital Systems

Barry W. Johnson

University of Virginia, Charlottesville

Addison-Wesley Publishing Company

Reading, Massachusetts • Menlo Park, California • New York
Don Mills, Ontario • Wokingham, England • Amsterdam • Bonn
Sydney • Singapore • Tokyo • Madrid • San Juan

4

Evaluation Techniques

4.1 Introduction
4.2 Quantitative Evaluation Methods
4.3 Reliability Modeling
4.4 Safety Modeling
4.5 System Comparisons
4.6 Availability Models
4.7 Maintainability Models
4.8 Redundancy Ratios
4.9 Qualitative Methods
4.10 Tradeoff Analysis Example
Summary
References
Additional Reading
Problems

4.1 Introduction

The techniques presented in the previous chapter form a collection of approaches that can be used to achieve fault tolerance. The specific techniques used in a given application depend on not only the application, but also on the ideas and philosophies of the designers. One team of designers might develop one fault-tolerant design, whereas another team would choose a completely different approach for the same application. It is important to be

6.7.5 Redundancy to Enhance Yield of VLSI Circuits 439
Summary 451
References 453
Additional Reading 454
Problems 457

7 Testing **463**

7.1 Introduction 463
7.2 Fault-Testing 466
7.3 Test Pattern Generation 468
7.3.1 Fault Tables 468
7.3.2 Adaptive Experiments 477
7.3.3 Boolean Differences 481
7.3.4 Literal Propositions 488
7.3.5 Path Sensitization 491
7.3.6 The D-Algorithm 498
7.3.7 Fault Simulation for Test Pattern Generation 510
The Row Method 513
The Column Method 513
7.4 Random Testing 516
7.5 Signature Analysis 517
7.6 Design for Testability 520
7.6.1 Scanning as a Method of Design for Testability 524
Level Sensitive Scan Design (LSSD) 525
Scan Path 529
Scan/Set Logic 532
Random-Access Scan 535
7.6.2 Sample Design: Comparing LSSD and Scan Path 538
7.6.3 Built-In Logic Block Observation 544
7.7 Testability Analysis 552
7.7.1 Important Definitions 553
7.7.2 Testability Analyzers 554
SCOAP 555
CAMELOT 559
7.7.3 Comparison of Testability Analyzers 562
7.7.4 Analysis of Circuits Containing Redundancy 564
Summary 565
References 568
Additional Reading 570
Problems 572

1 Introduction

1.1 Overview
1.2 Origins of Fault-Tolerant Computing
1.3 Goals of Fault Tolerance
1.4 Applications of Fault-Tolerant Computing
1.5 Fault Tolerance as a Design Objective
Summary
References
Additional Reading

1.1 Overview

This textbook is devoted to the study of techniques for designing and analyzing fault-tolerant and easily-testable systems. A **fault-tolerant system** is one that can continue to correctly perform its specified tasks in the presence of hardware failures and software errors. For example, the effect of a software "bug" in a fault-tolerant system is overcome so that the system continues correct operation. Likewise, the failure of a hardware component in a fault-tolerant system does not inhibit that system's ability to correctly execute its design-specified functions. **Fault tolerance** is the attribute that enables a system to achieve fault-tolerant operation. Finally, the term **fault-tolerant computing** describes the process of performing calculations, such as those performed by a computer, in a fault-tolerant manner.

An **easily-testable system** is one whose ability to perform correctly can be verified in a simple and straightforward manner. The complexity of

today's systems demands that special features be incorporated into the system to support testing. **Design for testability** is the process by which such features are included.

The concept of fault tolerance has become increasingly important during the past decade because of the increased use of computers in the vital aspects of almost everyone's life. Computers are no longer confined to use as powerful calculators where their incorrect performance can produce little more than frustration and lost time. Instead, computers are now integrated into commercial and military aircraft flight control systems ([Wensley et al. 1978], [Hopkins et al. 1978], and [Bosch and Kuehl 1977]), industrial controllers ([Ayache, Courtiat, and Diaz 1982] and [Wensley and Harclerode 1982]), space applications [Rennels 1978], and banking systems ([Katzman 1977], [Manual 1982], and [Herbert 1983]). In each application, erroneous computer performance can be devastating to financial records, environmental safety, national security, and even human life. In summary, fault tolerance has become more important simply because the functions of computers and other digital systems have become more crucial.

This book discusses the many aspects of designing and analyzing fault-tolerant systems. In addition, design for testability methods and techniques are presented. Specific topics covered include:

1. Fundamental terminology crucial to the understanding of fault tolerance and design for testability
2. Techniques for designing fault-tolerant systems
3. The use of fault tolerance to achieve design goals such as reliability
4. Measures of the quality of a fault-tolerant design
5. Practical examples of fault-tolerant systems
6. The impact of integrated circuit technology on the design of fault-tolerant systems
7. Techniques of design for testability

We begin our discussions in this chapter by examining some of the historical aspects of fault-tolerant computing. Also, we consider in more detail the design goals that might be satisfied through the use of fault tolerance, as well as the role of fault tolerance in the design process. Several key concepts presented in this chapter include:

1. Definitions of reliability, availability, maintainability, safety, dependability, performability, and testability
2. The distinction between fault tolerance and reliability
3. The applications in which fault tolerance is most frequently used
4. The role of fault tolerance in the design process

1.2 Origins of Fault-Tolerant Computing

Fault tolerance is certainly not a new field. The first digital computers made extensive use of error detection and fault tolerance techniques to overcome the low reliability of their basic components [Carter and Bouricius 1971]. Some of the early Bell Relay Computers (BRC), for example, had two central processing units [ERA 1950]; one unit would begin executing the next instruction when the other unit encountered an error. Later versions of the BRC used a retry mechanism to repeat an operation immediately after an error was detected. The IBM 650, UNIVAC, and the Whirlwind I computers [Weik 1955] incorporated parity to check the results of data transfers. The EDVAC computer [Carter and Bouricius 1971], designed in 1949, is generally considered to have been the first computer to completely duplicate the Arithmetic Logic Unit (ALU) and compare the results obtained by each unit; the processing continued as long as the two ALUs agreed.

The advent of the transistor, along with its increased reliability, led to a temporary decrease in the emphasis on fault-tolerant computing. For many designers, the major thrust was to increase computer performance and speed and to depend on the improved reliability of the transistor to guarantee correct computations. It was not until computers began performing much more critical tasks that fault tolerance again surfaced as a crucial issue. Perhaps the best examples are in the United States space program and in military applications. The increase in computational requirements in many of these applications mandated the use of digital computers, and the significant penalties for incorrect performance required that the computers perform their functions without error. As an example, the IBM Saturn V system [Kuehn 1969] used triplicated modules and parity checking to improve the fault tolerance capability of the system.

The first theoretical work in fault-tolerant computing is generally credited to John von Neumann [von Neumann 1956]. In 1952, von Neumann presented a series of lectures on the use of replicated logic modules to improve system reliability. von Neumann later developed an article entitled "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components" [von Neumann 1956] in which he presented the concept of majority voting and analyzed the impact that such arrangements could have on the probability of a system producing erroneous results.

Since about 1970, the field of fault-tolerant computing has been rapidly developing. Several excellent journals such as *Computer*, *IEEE Micro*, the *Proceedings of the IEEE*, the *Journal of Design Automation and Fault-Tolerant Computing*, and the *IEEE Transactions on Computers* regularly present special issues that deal solely with fault-tolerant computing. In addition, the International Symposium on Fault-Tolerant Computing, commonly called

the Fault-Tolerant Computing Symposium (FTCS), has been held each year, beginning as the Symposium of Fault-Tolerant Computing in 1971.

Despite the apparent progress and the prolific dissemination of information, the field of fault-tolerant computing is still relatively immature, particularly when it comes to the application of the technology. Much of this should change, however, partly because of the advent of Very Large Scale Integration (VLSI). VLSI technology has made the implementation of many fault tolerance techniques not only feasible, but in many cases, extremely practical and cost effective. VLSI, however, introduces new problems in the design of fault-tolerant systems that previously did not have to be addressed. Consequently, the advantages of VLSI do not come to us free of charge. Subsequent chapters of this book address the relationship between fault tolerance and VLSI in significant detail.

1.3 Goals of Fault Tolerance

It is natural to ask at this point why fault tolerance is so important and why it is the concern of so many designers. Fault tolerance is an attribute that is designed into a system to achieve some design goal. Just as a design must meet many functional and performance goals, it must satisfy numerous other requirements as well. The most prominent of the additional requirements are reliability, availability, safety, performability, dependability, maintainability, and testability. Fault tolerance is one system attribute capable of fulfilling such requirements. This chapter provides an overview of each requirement; Chapter 4 describes techniques that allow the evaluation of each attribute.

1.3.1 Reliability

The **reliability** $R(t)$ of a system is a function of time, defined as the conditional probability that the system will perform correctly throughout the interval $[t_0, t]$, given that the system was performing correctly at time t_0. In other words, the reliability is the probability that the system will operate correctly throughout a complete interval of time. The reliability is a conditional probability in that it depends on the system being operational at the beginning of the chosen time interval. The **unreliability** $Q(t)$ of a system is a function of time, defined as the conditional probability that a system will perform *incorrectly* during the interval $[t_0, t]$, given that the system was performing *correctly* at time t_0. The unreliability is often referred to as the *probability of failure.*

Reliability is most often used to characterize systems in which even momentary periods of incorrect performance are unacceptable, or in which it is impossible to repair the system. If repair is impossible, such as in many space applications, the time intervals being considered can be extremely long, perhaps as many as ten years. In other applications, such as aircraft flight control, the time intervals of concern can be no more than several hours, but the probability of working correctly throughout that interval can be 0.9999999 or higher. It is a common convention when reporting reliability numbers to use 0.9_i to represent the fraction that has i nines to the right of the decimal point. For example, 0.9999999 is written as 0.9_7.

It is important to understand the difference between fault tolerance and reliability. Fault tolerance is a technique that can improve reliability, but a fault-tolerant system does not necessarily have a high reliability. A system can be designed to tolerate any single hardware failure or software error that can occur, but the probability of such problems existing can be so high that the reliability is very low. Likewise, a highly-reliable system is not necessarily fault tolerant. A very simple system might be designed using extremely good components such that the probability of the hardware failing is very low, but if the hardware does fail, the system cannot continue its functions. In other words, the system can achieve a high reliability but not possess the attribute of fault tolerance.

In summary, fault tolerance can improve a system's reliability by keeping the system operational when hardware failures and software errors occur. For example, a computing system that has redundant processors can often be designed to continue the correct performance of its tasks, even when one or more of the processors becomes inoperable.

1.3.2 Availability

Availability is another design goal that we can achieve through the use of fault tolerance. **Availability** $A(t)$ is a function of time, defined as the probability that a system is operating correctly and is available to perform its functions at the instant of time t. Availability differs from reliability in that reliability depends on an *interval* of time, whereas availability is taken at an *instant* of time. A system can be highly available yet experience frequent periods of inoperability as long as the length of each period is extremely short. In other words, the availability of a system depends not only on how frequently it becomes inoperable but also on how quickly it can be repaired. The most common measure of availability is the expected fraction of time that a system is available to correctly perform its functions.

Availability is most often used as a design goal when the system's primary purpose is to provide its services as often as possible. Examples of high-availability applications include time-shared computing systems and certain transactions processing applications, such as airline reservation systems. The users of highly-available systems want those systems to possess a

high probability of performing correctly at the instant they are requested to do so.

Fault tolerance offers numerous ways in which to improve the availability of a system. For example, the use of spare processors in a computing system can allow the functions of the system to be performed by a spare processor in the event that the primary processor becomes inoperable. In other words, the spare processor can perform the functions of the system while the primary processor is being repaired, thus keeping the system available for use.

1.3.3 Safety

One attribute that is often overlooked is the safety of a system. **Safety** $S(t)$ is the probability that a system will either perform its functions correctly or will discontinue its functions in a manner that does not disrupt the operation of other systems or compromise the safety of any people associated with the system. Safety is a measure of the *fail-safe* capability of a system; if the system does not operate correctly, you at least want the system to fail in a safe manner. For example, a pilot can safely fly an airplane, even if the autopilot fails, as long as the failure does not inhibit the aircraft's normal flight modes. Likewise, if a control valve for a chemical process fails, you often prefer that the valve fail in the closed position. Safety is the probability that these safe actions will result.

Safety and reliability differ because reliability is the probability that a system will perform its functions correctly, whereas safety is the probability that a system will either perform its functions correctly or will discontinue the functions in a manner that causes no harm. Certain techniques can be used to improve safety by turning a system off if a failure of some sort is detected. For example, in a nuclear power plant the reaction process should be stopped if some discrepancy is detected; this is the safe course of action.

1.3.4 Performability

In many cases, it is possible to design systems that can continue to perform correctly after the occurrence of hardware failures and software errors, but the *level* of performance is somehow diminished. For example, in today's era of multiprocessors, the failure of a single processor might not render the complete machine inoperable, but instead might simply decrease the speed of operation or decrease the amount of memory available to any one user. The multiprocessor, in this example, can still perform its tasks, but the relative quality of the performance has been decreased.

The **performability** $P(L, t)$ of a system is a function of time, defined as the probability that the system performance will be at, or above, some level L at the instant of time t [Fortes and Raghavendra 1984]. If we relate performability to the multiprocessor example, the level of performance might simply be the number of processors available for computational use. Performability differs from reliability in that reliability is a measure of the likelihood that *all* of the functions are performed correctly, whereas performability is a measure of the likelihood that some subset of the functions is performed correctly.

Graceful degradation is an important feature that is closely related to performability. **Graceful degradation** is the ability of a system to automatically decrease its level of performance to compensate for hardware failures and software errors. For example, if an airplane's autopilot begins to perform incorrectly, the graceful degradation might consist of simply disabling the autopilot. The level of performance would be that coinciding with the loss of the autopilot, and the performability would be the probability of being at that level of performance at time t. Fault tolerance can provide graceful degradation and improve performability by eliminating failed hardware and software from a system, thereby allowing performance at some reduced level.

1.3.5 Maintainability

Almost every design has maintainability as a goal. **Maintainability** is a measure of the ease with which a system can be repaired, once it has failed. In more quantitative terms, maintainability $M(t)$ is the probability that a failed system will be restored to an operational state within a specified period of time t. The restoration process includes locating the problem, physically repairing the system, and bringing the system back to its operational condition. Maintainability is crucial in all systems, but it is particularly important when human lives, equipment, or the environment are placed in jeopardy while a system is repaired.

Many of the techniques that are so vital to the achievement of fault tolerance can be used to detect and locate problems in a system for the purpose of maintenance. Once the problem is located, maintenance personnel can then perform the necessary repairs. Automatic diagnostics can significantly improve the maintainability of a system because a majority of the time used to repair a system is often devoted to determining the source of the problem.

1.3.6 Testability

A **test** is a means by which the existence and quality of certain attributes within a system are determined. For example, if a computer is supposed to

execute one million instructions per second, you would probably want to design a test to verify that the computer could indeed run at that particular rate. **Testability** is the *ability* to test for certain attributes within a system. Measures of testability allow us to assess the ease with which certain tests can be performed. As we will discover in subsequent chapters, certain tests can be automated and provided as an integral part of the system to improve the testability. Many of the techniques that are so vital to achieving fault tolerance can be used to detect and locate problems in a system for the purpose of improving testability. Testability is clearly related to maintainability because it is important to minimize the time required to identify and locate specific problems.

1.3.7 Dependability

The term **dependability** encompasses the concepts of reliability, availability, safety, maintainability, performability, and testability. Dependability is the quality of service that a particular system provides [Laprie 1985]. Reliability, availability, safety, maintainability, performability, and testability are measures used to quantify the dependability of a system.

1.4 Applications of Fault-Tolerant Computing

The use of fault-tolerant computing has spread into a number of fields for several reasons. First, a better understanding of fault tolerance techniques exists today. The field has grown from a handful of researchers twenty-five years ago to the point where several companies concentrate solely on fault-tolerant systems. Examples include Tandem Computers, Stratus Computers, and August Systems. Both Tandem and Stratus are competitors in the transactions processing industry, whereas August Systems concentrates on the development of reliable and fault-tolerant systems for industrial process control. Second, the advent of Large Scale Integration (LSI) and Very Large Scale Integration (VLSI) has made many fault tolerance techniques practical for the first time. Previously, designers could barely get a single computer to fit within their size, weight, and power consumption budgets. Fault-tolerant systems that used multiple computers were simply not practical in any but the most crucial applications. Finally, many systems that were previously mechanical are now electronic because of the tremendous capability and flexibility offered by electronics. Fault tolerance is now required because the new electronic systems are often less reliable.

Existing applications of fault-tolerant computing can be categorized into four primary areas: long-life applications, critical computations, maintenance postponement, and high availability. Each application presents differing design requirements and challenges.

1.4.1 Long-Life Applications

The most common examples of **long-life applications** are the unmanned space flight and satellites. The *Pioneer 10* spacecraft, for example, was launched on March 2, 1972, and became the first man-made object to pass beyond all known planets on June 13, 1983 [Lerner 1983]. During *Pioneer 10's* flight it has returned fascinating pictures of Jupiter and its moons, among other things. The information returned by *Pioneer 10* would have been severely restricted had not the electronics continued to function correctly throughout the time required for the spacecraft to reach its destination and perform its functions. The Mariner, Explorer, and Voyager missions are other examples of long-life space missions.

Satellites are also required to function correctly in space for extended periods of time. The cost of designing, building, and launching a satellite is much too high to allow electronic failures to render the satellite ineffective in space. Even though the space shuttle is now capable of retrieving satellites for repair, the cost of such repair is still extremely high, and many satellites are in orbits beyond the reach of the shuttle. Consequently, fault tolerance is required in satellite systems.

Typical requirements of a long-life application are to have a 0.95 probability of being operational at the end of a ten-year period. Unlike other applications, however, long-life systems can often allow extended outages as long as the system can eventually be made operational once again. For example, a one-week outage can be insignificant when you consider the five- or ten-year operational life of a satellite. In addition, long-life applications can frequently allow the system to be reconfigured manually by the operators. The Fault-Tolerant Spaceborne Computer (FTSC) [Stiffler 1976], the Self-Testing And Repairing (STAR) computer [Avizienis et al. 1971], and the Fault-Tolerant Building Block Computer (FTBBC) [Rennels 1980] are examples of systems designed for long-life applications.

As an example of a fault-tolerant computer system intended for long-life applications, Fig. 1.1 shows a general block diagram of the electronics found on board the *Voyager* spacecraft ([Jones 1979] and [Pradhan 1986]). The system consists of eight primary elements including the flight data system, the attitude control system, the command and control system, the radio system, a telemetry modulator, a command detector, a receiver, and a tape recorder. Two identical copies of each element are provided: one copy, called the *primary* copy, performs all the operations under normal circumstances and the second copy serves as a *backup*. The system is designed so

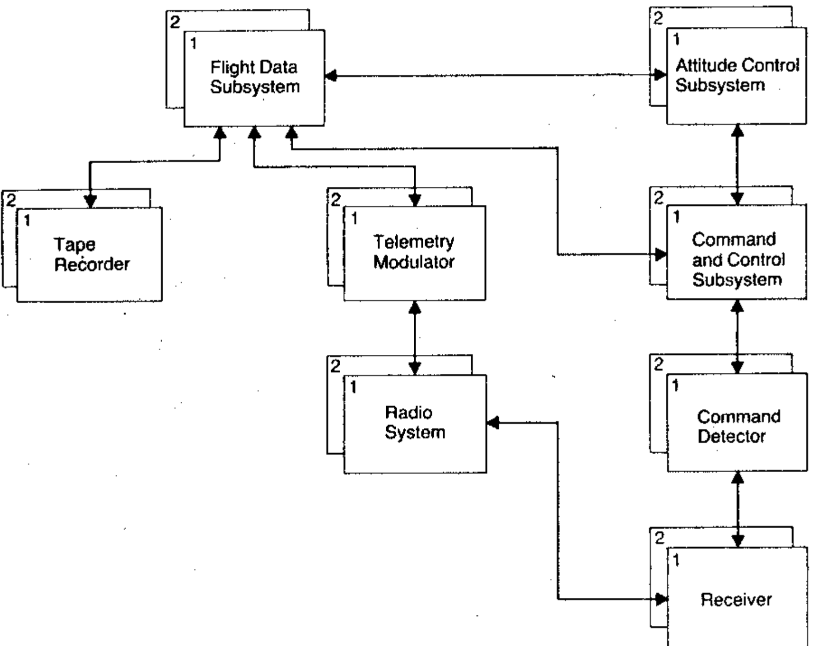


Fig. 1.1 The electronics on board the *Voyager* spacecraft achieve fault tolerance by using two identical copies of each major element.

that if the primary element fails, the backup element can be "switched in" to replace the primary and keep the system operational.

1.4.2 Critical-Computation Applications

Perhaps the most widely publicized applications of fault-tolerant computing are those in which the computations are critical to human safety, environmental cleanliness, or equipment protection. Examples include aircraft flight control systems, military systems, and certain types of industrial controllers. **In critical-computation applications,** the incorrect performance of the system will almost certainly yield devastating results. A typical requirement for a critical-computation application is to have a reliability of 0.9_7 at

the end of a three-hour period. Requirements can vary, however, depending on the particular function that the system is performing.

The most publicly visible critical-computation application of fault-tolerant computing has been the space shuttle. A malfunction in the shuttle's flight control system during either ascent or descent can result in the loss of the shuttle. Consequently, extreme care has been taken to ensure that the system performs its tasks dependably. In fact, the shuttle can continue its mandatory flight control functions after as many as three computer failures [Sklaroff 1976].

Industrial control systems also perform critical computations. For example, chemical reactions may have to be precisely controlled to prevent explosions or other unwanted effects. The goal in almost all critical-computation applications is to prevent the electronics from being the *weak point* in the system; fault tolerance is a means of accomplishing this design goal.

As an example of a fault-tolerant system used in a critical-computation application, consider the architecture of the X-29 aircraft flight control system, as shown in Fig. 1.2 [Anderson 1983]. The forward swept wing technology that the X-29 uses to maximize aerodynamic benefits requires a stabilizing control system. If the control system fails to perform correctly, the airplane will not be flyable. A hybrid analog/digital fly-by-wire flight control system is used to provide closed-loop control of the aircraft. The term *fly-by-wire* simply means that there are no mechanical connections between the pilot's stick and the control surfaces (for example, the ailerons, elevators, and rudder). Instead, an electronic system samples the position of the pilot's stick, calculates the desired position of the control surfaces, and commands a motor to move the control surfaces. The connection between the pilot's stick and the control surfaces is completely electrical; consequently, the loss of the electronics implies the loss of the ability to fly the aircraft.

As shown in Fig. 1.2, the control system uses three identical computers performing the same operations. The results from each computer are examined, and the output from the system is formed via a majority vote of the three results. Consequently, a single computer performing incorrectly will be overruled by the two computers that are performing correctly. Each computer within the system consists of both a digital computer and an analog computer. The analog computer is used as a backup that can assume the functions of the system if the digital computer fails. The analog backup provides protection against software errors that could simultaneously affect all the digital computers.

1.4.3 Maintenance Postponement Applications

Maintenance postponement applications appear most frequently when maintenance operations are extremely costly, inconvenient, or difficult to

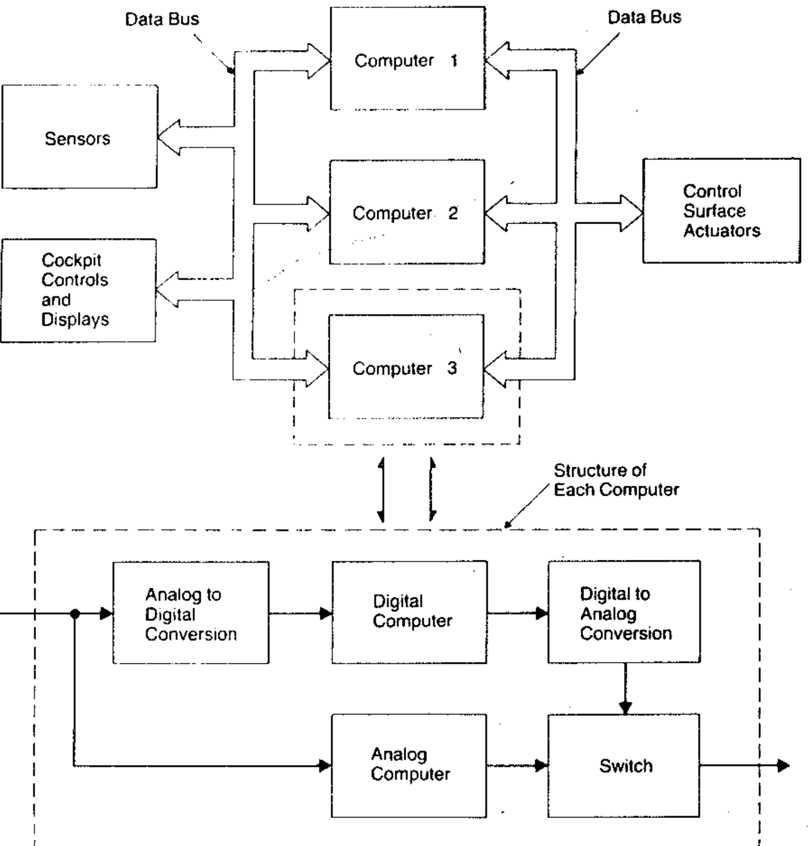


Fig. 1.2 The X 29 flight control system contains three complete computer systems. In addition, each computer system contains both an analog and a digital computer.

perform. Remote processing stations and certain space applications are good examples. In space, maintenance can be impossible to perform; at remote sites, the cost of unexpected maintenance can be prohibitive. The main goal is to use fault tolerance to allow maintenance to be postponed until a more convenient and cost-effective time. Maintenance personnel can visit a site monthly and perform any necessary repairs. Between maintenance visits, the system uses fault tolerance to continue to perform its tasks.

A telephone switching system [Toy 1978] is an example of a system that could require maintenance postponement. Many telephone switching systems are located in remote areas where it is necessary to provide telephone service, but it is costly to perform the maintenance and service operations. The primary objective is to design the system such that unscheduled maintenance can be avoided. Therefore, the telephone company can visit the facility periodically and repair the system or perform routine maintenance. Between maintenance visits, the system handles failures and service disruptions autonomously.

Figure 1.3 shows the block diagram of the 3B20D processor used in the Electronic Switching System (ESS) developed by AT&T Bell Laboratories [Serlin 1984]. Each element of this system is completely duplicated. One set of elements can be used to perform all the system functions, whereas the duplicate set serves as a backup in the event of a hardware failure. The duplicate set of elements can allow the system to remain functional while waiting for a repair to occur. Note in Fig. 1.3, for example, that either processor can access either storage disk, so the failure of a disk does not render the system inoperable.

1.4.4 High-Availability Applications

Availability is rapidly becoming a key parameter in many applications. Banking and other time-shared systems are good examples of **high-**

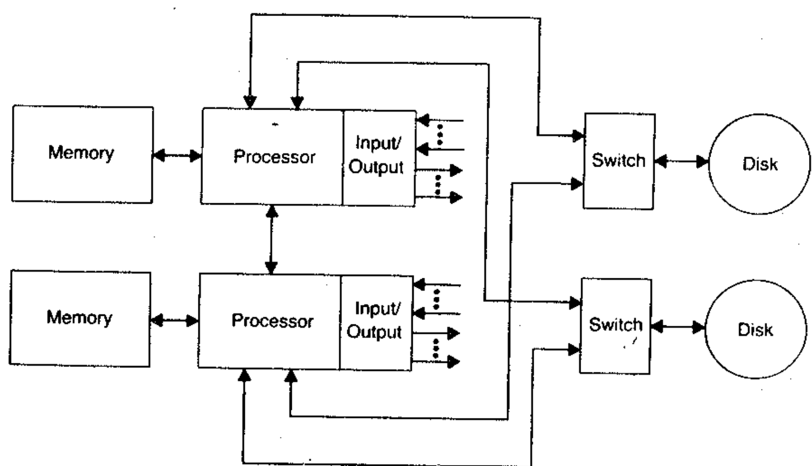


Fig. 1.3 Block diagram of the 3B20D processor used in the Bell Electronic Switching System. All critical components in this system are duplicated. (From [Serlin 1984] © 1984 IEEE)

availability applications. Users of these systems want to have a high probability of receiving service when it is requested. The Tandem NonStop transaction processing system [Katzman 1977] is a good example of one designed for high availability. A major competitor of the Tandem computer is the Stratus system [Herbert 1983]. Both the Tandem and the Stratus computers are designed to achieve a high probability of being operational when their services are required.

Intel's 432 processor system [Siewiorek 1982] is an example design developed to support high availability in many general-purpose processing applications. Intel's 432 system employs a number of techniques that support fault-tolerant operation. For example, Fig. 1.4 shows the structure of the central processing unit (CPU) and illustrates how two CPUs can be operated as a pair. If CPU 1 in Fig. 1.4 is enabled, the system's outputs will come from CPU 1, and CPU 2 will check those outputs with its own. Similarly, CPU 2 could be enabled, and CPU 1 would serve as the checker.

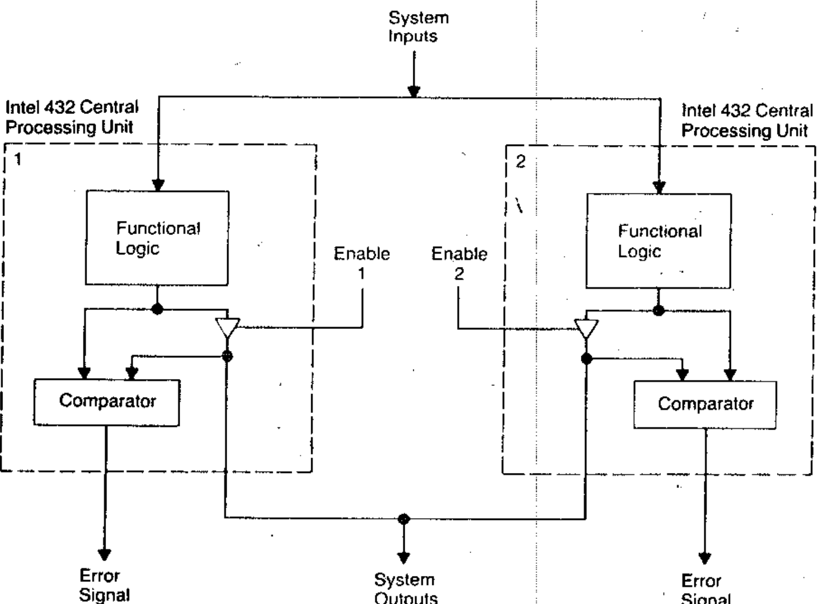


Fig. 1.4 The Intel 432 Central Processing Unit (CPU) supports operating two CPUs as a pair. The outputs of each CPU are compared. (From [Serlin 1984] © 1984 IEEE)

1.5 Fault Tolerance as a Design Objective

The design of a system is often conducted with many goals in mind. Certainly, we want the system to perform its intended function, but we also want the system to be dependable, cost effective, efficient, easy to test, and easy to repair. To accomplish our design goals, we begin with specific requirements. We may mandate that the cost and power not exceed certain values. Likewise, our design must meet specific reliability, availability, or maintainability goals. Fault tolerance plays an important role in attaining these goals, but it is not the only key part of the design process. Fault tolerance is a means of achieving our goals, but it must be coupled with other design techniques to be successful.

Figure 1.5 shows a top-level view of the design process. The system requirements, such as reliability, are achieved through two primary means: system design and system evaluation. The system design includes both fault avoidance and fault tolerance techniques. **Fault avoidance** techniques are performed to help prevent hardware failures and software errors. Examples include selecting high-quality components, enforcing design rules, and reviewing the designs periodically. Fault tolerance techniques, on the other hand, handle hardware failures and software errors when they occur. Examples include the use of redundant hardware, voting, and reconfiguration techniques.

The evaluation of a system is often overlooked as an integral part of the design process. Evaluation must be used in parallel with the design process, if the design is to be successful. System evaluation can uncover problems with a design early enough to allow corrections to be implemented. For example, if a design problem is discovered before the design is committed to hardware, the problem usually can be corrected easily. However, if problems remain in a system after it is built, correction can be impossible, or significant performance degradations may have to be accepted to allow the correction to be made. Numerous evaluation methods are available to analyze systems. Examples include Markov reliability models, system repair models, combinatorial reliability models, availability models, and maintainability models. In addition, evaluation techniques allow us to locate areas within a system that are prone to failure or where failure can be catastrophic. Each evaluation technique is crucial to a quality system design.

The primary purpose of this textbook is to study fault avoidance, fault tolerance, and system evaluation techniques that can be used in the design and analysis of fault-tolerant systems. This book presents the techniques that are available to achieve fault tolerance. More importantly, however, this book shows practical examples of how fault tolerance can be used to achieve design goals such as reliability, availability, and maintainability.

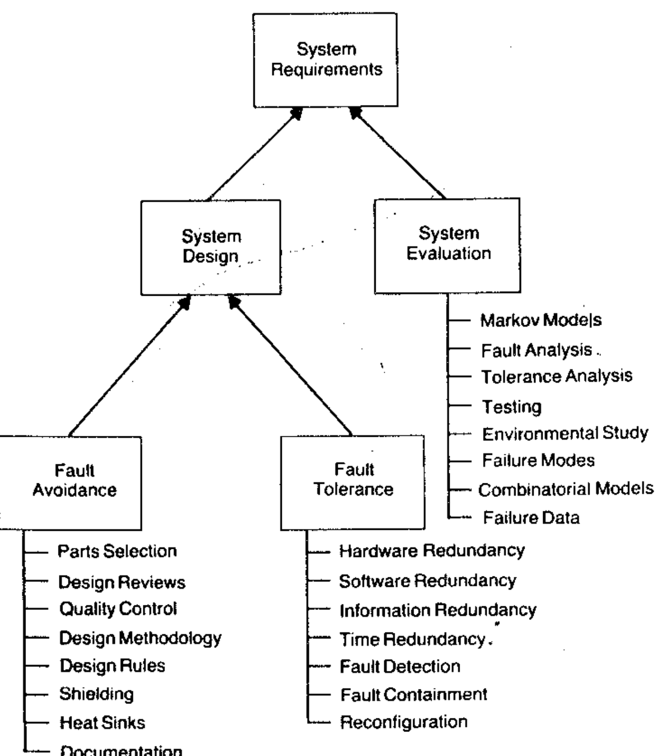


Fig. 1.5 A top-level view of the system design process illustrating the importance of fault avoidance, fault tolerance, and system evaluation.

Summary

This chapter has introduced the basic concept of fault tolerance and has illustrated the design goals often achieved via fault tolerance. The applications that require fault tolerance have been discussed, and examples from each application area have been presented. Perhaps the most important concept presented in this chapter is that fault tolerance is one aspect of a system design. To achieve design goals such as reliability or availability, a designer must use not only fault tolerance but fault avoidance and system evaluation as well.

The following list summarizes the important terminology and concepts that have been presented in this chapter.

Availability $A(t)$—the probability that a system is operating correctly and is available to perform its functions at the instant of time t.

Critical-Computation Application—an application in which the incorrect performance of computations can create devastating results.

Dependability—the quality of service provided by a particular system.

Design for Testability—the process of including special features to make a system easily testable.

Easily-Testable System—a system whose ability to perform correctly can be verified in a simple and straight forward manner.

Fault Avoidance—the process of attempting to prevent hardware failures and software errors from occurring in a system.

Fault-Tolerant Computing—the process of performing calculations, such as those performed by a computer, in a fault-tolerant manner.

Fault-Tolerant System—a system that can continue the correct performance of its specified tasks in the presence of hardware failures and software errors.

Fault Tolerance—the quality or attribute that enables a system to behave in a fault-tolerant manner.

Graceful Degradation—the ability of a system to automatically decrease its level of performance to compensate for hardware failures and software errors.

High-Availability Application—an application in which availability is the crucial design requirement.

Long-Life Application—an application in which the longevity of operation is the crucial design requirement.

Maintainability, $M(t)$—the probability that an inoperable system will be restored to an operational state within the time t.

Maintenance Postponement Application—an application in which it is desired to delay the process of repairing a system until the most convenient times.

Performability, $P(L,t)$—the probability that a system is performing at or above some level of performance L at the instant of time t.

Reliability, $R(t)$—the conditional probability that a system performs correctly throughout an interval of time $[t_0, t]$, given that the system was performing correctly at time t_0.

Safety, $S(t)$—the probability that a system will either perform its functions correctly or will discontinue its functions in a well-defined, safe manner.

Test—a means by which the existence and quality of certain attributes within a system is determined.

Testability—the ability to test for certain attributes within a system.

Unreliability, $Q(t)$—the conditional probability that a system will perform *incorrectly* during the interval of time $[t_0, t]$, given that the system was performing *correctly* at time t_0.

References

1. Anderson, D. "X-29 Forward swept wing flight control system," *Proceedings of the Joint AIAA-IEEE Fifth Digital Avionics Systems Conference*, Washington, D.C., December 1984, pp. 1–8.
2. Avizienis, A., Gilley, G.C., F.P. Mathur, D.A. Rennels, J.A. Rohr, and D.K. Rubin. "The STAR (Self-Testing And Repairing) computer: An investigation of the theory and practice of fault-tolerant computer design," *IEEE Transactions on Computers*, Vol. C-20, No. 11, November 1971, pp. 1312–1321.
3. Ayache, J., J. Courtiat, and M. Diaz. "REBUS: A fault tolerant distributed system for industrial real-time control," *IEEE Transactions on Computers*, Vol. C-31, No. 7, July 1982, pp. 637–647.
4. Bosch, J.A., and W.J. Kuehl. "Reconfigurable redundancy management for aircraft flight control," *Journal of Aircraft*, Vol. 14, No. 10, October 1977, pp. 966–971.
5. Carter, W.C., and W.G. Bouricius. "A survey of fault tolerant computer architecture and its evaluation," *Computer*, Vol. 4, No. 1, January 1971, pp. 9–16.
6. Engineering Research Association. *High Speed Computing Devices*, McGraw-Hill, 1950.
7. Fortes, J.A.B., and C.S. Raghavendra. "Dynamically reconfigurable fault-tolerant array processors," *Proceedings of the 14th International Conference on Fault-Tolerant Computing*, Kissimmee, Fla., June 20–22, 1984, pp. 386–392.
8. Herbert, E. "Computers: Minis and mainframes," *IEEE Spectrum*, Vol. 20, No. 1, January 1983, pp. 28–33.
9. Hopkins, A.L., T.B. Smith, and J.H. Lala. "FTMP: A highly reliable fault-tolerant multiprocessor for aircraft," *Proceedings of the IEEE*, Vol. 66, No. 10, October 1978, pp. 1221–1239.
10. Jones, C.P. "Automatic fault protection in the Voyager spacecraft," AIAA Paper No. 79–1919, American Institute of Aeronautics and Astronautics.
11. Katzman, J.A. "System architecture for nonstop computing," *Proceedings of the 14th Computer Society International Conference (Compcon)*, San Francisco, February 1977, pp. 77–80.

12. Kuehn, R.E. "Computer redundancy: Design, performance, and future," *IEEE Transactions on Reliability*, Vol. R-18, No. 1, February 1969, pp. 3–11.
13. Laprie, J.C. Dependable computing and fault tolerance: Concepts and terminology, *Proceedings of the 15th Annual International Symposium on Fault-Tolerant Computing*, June 19–21, 1985, Ann Arbor, Michigan, pp. 2–11.
14. Lerner, E.J. "Crossroads in space," *Spectrum*, Vol. 20, No. 9, September 1983, pp. 28–55.
15. Manual, T. "New architecture cuts redundancies in fail-safe processing," *Electronics*, Vol. 55, No. 17, August 25, 1982, pp. 45–46.
16. Pradhan, D.K. *Fault-Tolerant Computing Theory and Techniques*, Prentice-Hall, Englewood Cliffs, N.J., 1986.
17. Rennels, D.A. "Architectures for fault tolerant spacecraft computers," *Proceedings of the IEEE*, Vol. 66, No. 10, October 1978, pp. 1255–1268.
18. Rennels, D.A. "Distributed fault-tolerant computer systems," *IEEE Computer*, Vol. 13, No. 3, March 1980, pp. 55–64.
19. Serlin, O. "Fault-tolerant systems in commercial applications," *Computer*, Vol. 17, No. 8, August 1984, pp. 19–30.
20. Siewiorek, D.P., and R.S. Swarz. *The Theory and Practice of Reliable System Design*, Digital Press, Bedford, Mass., 1982.
21. Sklaroff, J.R. "Redundancy management technique for the space shuttle computers," *IBM Journal of Research and Development*, Vol. 20, No. 1, January 1976, pp. 20–28.
22. Stiffler, J.J. "Architectural design for near-100% fault coverage," *Proceedings of the International Symposium on Fault Tolerant Computing*, 1976, pp. 134–137.
23. Toy, W.N. "Fault-tolerant design of local ESS processor," *Proceedings of the IEEE*, Vol. 66, No. 10, October 1978, pp. 1126–1145.
24. von Neumann, J. "Probabilistic logics and the synthesis of reliable organisms from unreliable components," *Automata Studies, Annals of Mathematical Studies*, Princeton University Press, No. 34, pp. 43–98, 1956.
25. Weik, M.H. "A survey of domestic electronic digital computing systems," Report #971, Commerce Department, Ballistic Research Laboratories, Aberdeen Proving Grounds, Md., December 1955.
26. Wensley, J.H., L. Lamport, J. Goldberg, M.W. Green, K.N. Levitt, P.M. Melliar-Smith, R.E. Shostak, and C.B. Weinstock, "SIFT: Design and analysis of a fault tolerant computer for aircraft control," *Proceedings of the IEEE*, Vol. 66, No. 10, October 1978, pp. 1240–1255.
27. Wensley, J.H., and C.S. Harclerode. "Programmable control of a chemical reactor using a fault tolerant computer," *IEEE Transactions on Industrial Electronics*, Vol. IE-29, No. 4, November 1982, pp. 258–264.

Additional Reading

For the reader interested in learning more about the history and development of fault-tolerant computing, the following list of suggested references is provided. This list primarily includes tutorial and survey articles that give the reader a good feel for the development of the technology, terminology, and applications of fault-tolerant computing.

Avizienis, A. "Fault tolerance: The survival attribute of digital systems," *Proceedings of the IEEE*, Vol. 66, No. 10, October 1978, pp. 1109–1125.

Avizienis, A. "Fault tolerant computing: An overview," *Computer*, Vol. 4, No. 1, January 1971, pp. 5–8.

Avizienis, A. "Architecture of fault-tolerant computing systems," *Proceedings of the 1975 International Symposium on Fault-Tolerant Computing*, Paris, June 1975, pp. 3–16.

Baechler, D.O. "Aerospace computer characteristics and design trends," *Computer*, Vol. 4, No. 1, January/February 1971, pp. 45–57.

Cooper, A.E., and W.T. Chow. "Development of on-board space computer systems," *IBM Journal of Research and Development*, Vol. 20, No. 1, January 1976, pp. 5–19.

Deyst, J.J., Jr., J.V. Harrison, E. Gai, and K. C. Daly. "Fault detection, identification, and reconfiguration for spacecraft systems," *Journal of Astronautical Science*, Vol. 29, No. 2, April-June 1981, pp. 113–126.

Goldberg, J. "New problems in fault-tolerant computing," *Proceedings of the 1975 International Symposium on Fault-Tolerant Computing*, Paris, 1975, pp. 29–34.

Harris, R.L., and E.E. Jones. "Fault tolerance applications to future military system avionics," *Proceedings of the IEEE 1980 National Aerospace and Electronics Conference*, Dayton, Oh., May 1980.

Hecht, H. "Fault tolerant computers for spacecraft," *Journal of Spacecraft*, Vol. 14, No. 10, October 1977.

Hopkins, A.L., Jr. "Design foundations for survivable, integrated, on-board computation and control," *Proceedings of the 1977 Joint Automatic Control Conference*, San Francisco, Calif., June 22–24, 1977, pp. 232–237.

Hopkins, A.L., Jr. "Fault tolerant system design: Broad brush and fine print," *Computer*, Vol. 13, No. 3, March 1980, pp. 39–46.

Jennings, R. "Fault secure avionic system development," *Proceedings of the 1981 International Aerospace and Electronics Conference*, Vol. 1, Dayton, Oh., May 9–11, 1981.

Kime, C.R. "Fault tolerant computing: An introduction and a perspective," *IEEE Transactions on Computers*, Vol. C-24, No. 5, May 1975, pp. 457–460.

Koczela, L.J., and G.J. Burnett. "Advanced space missions and computer systems," *IEEE Transactions on Aerospace and Electronic Systems*, Vol. AES-4, No. 3, May 1968, pp. 456–467.

Nelson, V.P., and B.D. Carroll. *Tutorial: Fault-Tolerant Computing*, IEEE Computer Society Press, Washington, D.C., 1986.

Ramamoorthy, C.V. "Fault tolerant computing: An introduction and an overview," *IEEE Transactions on Computers*, Vol. C-20, No. 11, November 1971, pp. 1241–1244.

Rennels, D.A. "Fault-tolerant computing—Concepts and examples," *IEEE Transactions on Computers*, Vol. C-33, No. 12, December 1984, pp. 1116–1129.

Rennels, D.A. "Reconfigurable modular computer networks for spacecraft on-board processing," *Computer*, Vol. 11, No. 7, July 1978, pp. 49–59.

Short, R., and J. Goldberg. "A summary of Soviet activities in the design of fault tolerant digital machines," *Computer*, Vol. 11, No. 1, January/February 1971, pp. 28–33.

Short, R., and J. Goldberg. "A survey of Soviet activities in the design of fault tolerant digital machines," *IEEE Transactions on Computers*, Vol. C-20, No. 11, November 1971, pp. 1337–1352.

Siewiorek, D.P., D.E. Thomas, and D.L. Scharfetter. "The use of LSI modules in computer structures: Trends and limitations," *Computer*, Vol. 11, No. 7, July 1978, pp. 16–25.

Teschler, L. "Computers that won't fail," *Machine Design*, Vol. 50, No. 10, May 11, 1978, pp. 91–97.

2 Fundamental Definitions

- **2.1** Introduction
- **2.2** Faults, Errors, and Failures
- **2.3** Causes of Faults
- **2.4** Characteristics of Faults
- **2.5** Fault Models
- **2.6** Error Models
- **2.7** Design Philosophies to Combat Faults
- Summary
- References
- Additional Reading
- Problems

2.1 Introduction

Throughout the history of fault-tolerant computing, there has been substantial disagreement on the definitions of several key concepts. For example, the terms *fault*, *failure*, and *error* have often been used interchangeably in the literature. To many people, a failure has occurred when the time-shared computer they are using fails to respond to their requests or demands. To other people, a failure is a more specific physical defect within some electronic component. Some groups view the physical defects as faults instead of failures.

The purpose of this chapter is to introduce the basic terminology used in the fault-tolerant computing field. It is important to understand the causes of faults and the types of faults that can occur before considering the

techniques that are available to tolerate faults in a design. In addition, it is vital to have a clear understanding of how faults manifest themselves in digital systems. In other words, once a fault has occurred, how is it likely to propagate throughout the system and what are the probable impacts of that fault? Finally, we discuss the role of fault tolerance and fault avoidance in combating faults in digital systems. Perhaps the best available references on the fault tolerance terminology are [Avizienis 1982], [Laprie 1985], and [Johnson 1984].

2.2 Faults, Errors, and Failures

Three fundamental terms in fault-tolerant design are fault, error, and failure. There is a cause-and-effect relationship between faults, errors, and failures. Specifically, faults are the cause of errors, and errors are the cause of failures.

A **fault** is a physical defect, imperfection, or flaw that occurs within some hardware or software component. Essentially, the definition of a fault, as used in the fault tolerance community, agrees with the definition found in the dictionary. A fault is a blemish, weakness, or shortcoming of a particular hardware or software component. Examples of faults include shorts between electrical conductors, opens or breaks in conductors, or physical flaws or imperfections in semiconductor devices. Similarly, in software, an example of a fault is a program loop that when entered can never be exited.

An **error** is the manifestation of a fault. Specifically, an error is a deviation from accuracy or correctness. For example, suppose that a physical short results in a line within a circuit being permanently stuck at a logic 1. The physical short is a fault within the circuit. If some condition occurs that requires the line to transition to a logic 0, the value on the line will be in error. In other words, the correct value for the line will be logic 0, but the existence of the fault has caused the line to have an erroneous value. In other words, an error is the result of a fault.

Finally, if the error results in the system performing one of its functions incorrectly, a system failure has occurred. Essentially, a **failure** is the nonperformance of some action that is due or expected. Although it is often used interchangeably with the term **malfunction,** the term *failure* is rapidly becoming more commonly accepted. A failure is also the performance of some function in a subnormal quantity or quality. As an example, suppose that a line in a circuit is responsible for turning a valve on or off: a logic 1 turns the valve on and a logic 0 turns the valve off. If the line is stuck at logic 1, the valve is stuck on. As long as the user of the system wants the valve on, the system will be functioning correctly. However, when the user wants to turn the valve off, the system will experience a failure.

Figure 2.1 illustrates the cause-and-effect relationship between faults, errors, and failures. Faults result in errors, and errors can lead to system failures. One way to think of Fig. 2.1 is as a hierarchy. At the bottom of the hierarchy are faults. Errors are the effect of faults, and, finally, failures are the effect of errors.

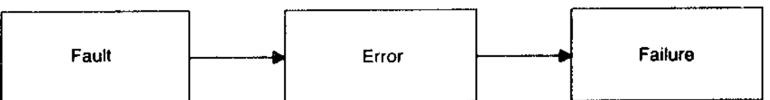


Fig. 2.1 Relationship between faults, errors, and failures. Failures are caused by errors, which are caused by faults.

The circuit shown in Fig. 2.2 further illustrates the distinction between faults and errors. The circuit of Fig. 2.2 is the logic diagram for a full-adder. The inputs A_1, B_1, and C_1 are the two bits of the operands and the carry bit, respectively. The truth table that shows the correct performance for this circuit is presented in Fig. 2.3. If a short occurs between line L and the power supply line resulting in line L becoming permanently fixed at a logic 1 value, a fault will have occurred. The fault is the actual short within the circuit.

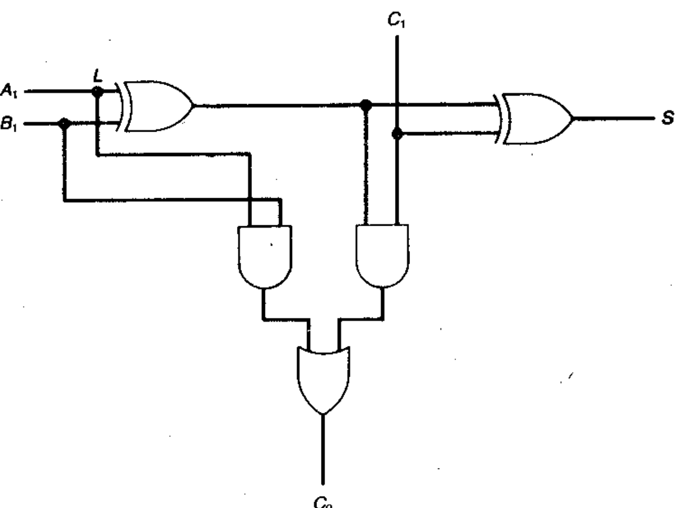
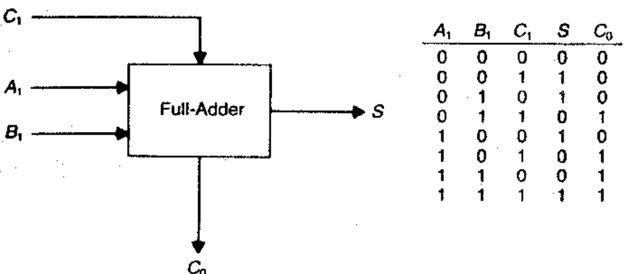


Fig. 2.2 Full-adder circuit to illustrate the difference between faults and errors.

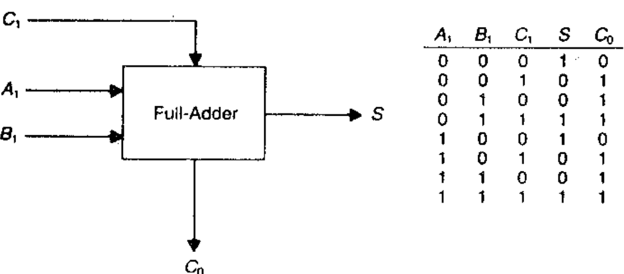


A_I	B_I	C_I	S	C_0
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Fig. 2.3 Truth table for the fault-free full-adder circuit.

Figure 2.4 shows the truth table of the circuit that contains the physical short. By comparing the truth tables in Figs. 2.3 and 2.4, we can see that the circuit performs correctly for the input combinations 100, 101, 110, and 111, but not for 000, 001, 010, and 011. The physical short within the circuit is a fault. Whenever an input is applied that results in the circuit producing an incorrect output, an error has occurred. If the output of the circuit is controlling a relay and the relay is opened when it should be closed, a failure has occurred.

The concepts of faults, errors, and failures can be best presented by using a three-universe model, which is an adaptation of the four-universe model originally developed in [Avizienis 1982]. The first universe is the **physical universe** in which faults occur. The physical universe contains the semiconductor devices, mechanical elements, displays, printers, power supplies, and other physical entities that make up a system. A fault is a physical defect or alteration of some component within the physical universe.



A_I	B_I	C_I	S	C_0
0	0	0	1	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Fig. 2.4 Truth table for the full-adder circuit with an example fault.

The second universe is the **informational universe.** The informational universe is where the error occurs. Errors affect units of information such as data words within a computer or digital voice or image information. Terms that are applicable to the informational universe include parity errors, message errors, typographical errors, and bit errors. An error has occurred when some unit of information becomes incorrect.

The third universe is the **external universe** (or **user's universe**). The external universe is where the user of a system ultimately sees the effect of faults and errors. The external universe is where failures occur. The failure is any deviation that occurs from the desired or expected behavior of a system. For example, the user may expect a system to correctly print payroll checks. If, for some reason, a check is printed incorrectly, the user has witnessed a failure of the system, and the failure has been witnessed in the external universe.

Figure 2.5 illustrates the relationships implied in the three-universe model. In summary, faults are physical events that occur in the physical universe. Faults can result in errors in the informational universe, and errors can ultimately lead to failures that are witnessed in the external universe of the system.

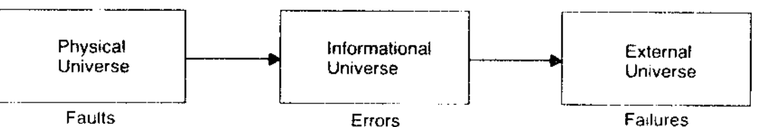


Fig. 2.5 Three-universe model representing the cause-and-effect relationship between faults, errors, and failures. Faults occur in the physical universe and cause errors to occur in the informational universe. Errors can result in failures that occur in the external universe.

The cause-and-effect relationship implied in the three-universe model leads to the definition of two important parameters: fault latency and error latency. **Fault latency** is the length of time between the occurrence of a fault and the appearance of an error due to that fault. A **latent fault** is one that is present in a system but has not yet produced an error. In other words, a latent fault has not yet produced any effect. **Error latency** is the length of time between the occurrence of an error and the appearance of the resulting failure. Based on the three-universe model, the total time between the occurrence of a physical fault and the appearance of a failure is the sum of the fault latency and the error latency.

2.3 Causes of Faults

Faults can be the result of a variety of things that occur within electronic components, external to the components, or during the component or system design process. It is very important to understand all the possible causes of faults. To understand the various causes of faults, we first examine a typical design process to identify areas where faults can occur.

A design process often begins with a somewhat vague statement of the problem. The designers generate the problem statement as their interpretation of the actual problem. From the problem statement, a high-level solution to the problem is outlined, and the development of specific algorithms and system architectures is begun. Hardware and software specifications are then developed from the initial algorithms and architectures. A complete hardware and software design and test procedure is then invoked.

Once the hardware and software are completed, the designs must pass through a complete integration and test procedure with the end result being an operational system that, designers hope, solves the original problem. The various parts of the design process are typically performed several times before the design is completed. For example, the preliminary design process can modify the problem statement, either by necessity or as the result of some design tradeoff.

Problems at any of several points within the design process can result in faults within the system. Possible **fault causes** can be associated with problems in four basic areas: specifications, implementation, components, and external factors.

The first cause of faults is the possibility of **specification mistakes.** These include incorrect algorithms, architectures, or hardware and software design specifications. For example, suppose that the designer of a digital circuit incorrectly specified the timing characteristics of some of the circuit's components. The design might perform correctly at times, but there could also be instances of incorrect performance.

The next cause of faults is **implementation mistakes.** Implementation, as defined here, is the process of transforming hardware and software specifications into the physical hardware and the actual software. The implementation can introduce faults due to poor design, poor component selection, poor construction, or software coding mistakes. For example, suppose that a printed circuit board is constructed such that adjacent lines of a circuit are shorted together. The components on the board can be performing correctly, but the board produces incorrect results because of a wiring mistake. A second crucial example of an implementation mistake is a software coding error. If the software is written incorrectly, a fault exists, and an error can result if the faulty software is executed at some point during the operation of the system.

The next cause of faults is **component defects.** Manufacturing imperfections, random device defects, and component wear-out are typical examples of component defects. Electronic components simply become defective sometimes. The defect can be the result of bonds breaking within the circuit or corrosion of the metal. Component defects are the most commonly considered cause of faults, but, as is evident from our discussions, component defects are only one of several causes of faults.

The final cause of faults is the **external disturbance,** for example, radiation, electromagnetic interference, battle damage, operator mistakes, and environmental extremes. If an electronic system is subjected to extreme temperature variations, the system can produce incorrect results. If the electronics in a military system are damaged during a battlefield encounter, a fault has been injected into the system by some external source. Also, electronic systems are usually very sensitive to electrostatic sources such as lightning or other weather-related effects. Finally, operator mistakes are considered to be external disturbances since the operator is external to the system's physical hardware and software. Clearly, an operator can provide incorrect commands to a system that can ultimately lead to system failures.

Figure 2.6 illustrates the general effects of faults in a system and follows the three-universe concepts discussed in the previous section. The four distinct causes—specification mistakes, implementation mistakes, component defects and external disturbances—result in hardware or software faults in the physical universe. The effect of a fault is to produce some error that is an unintentional deviation from correctness in the information of a system. The result of an error is a failure that occurs in the external universe.

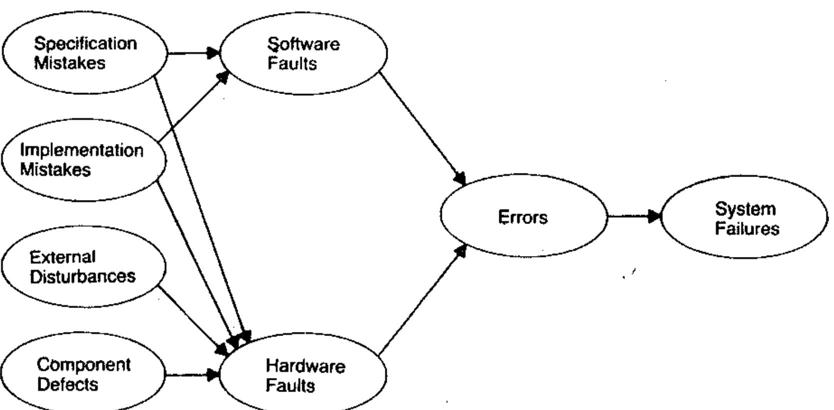


Fig. 2.6 Cause-and-effect relationship of faults, errors, and failures in a system.

2.4 Characteristics of Faults

We have spent quite some time developing the causes of faults and how faults, errors, and failures differ. To adequately describe faults, however, characteristics other than the cause are required. In addition to the cause, four major attributes are critical to the description of faults: nature, duration, extent, and value [Nelson and Carroll 1982]. Figure 2.7 illustrates each of the basic characteristics of faults.

The **fault nature** specifies the type of fault. For example, is the fault a hardware fault or a software fault? Also, is the fault in the analog or the digital circuitry? A power supply fault is an example of an analog hardware fault. A short circuit within a microprocessor is an example of a digital hardware fault. A loop within a subroutine that, when entered, can never be exited is an example of a software fault. In summary, the nature of a fault can be specified using the terms hardware, software, analog, and digital.

The **fault duration** specifies the length of time that a fault is active. First, there is the **permanent fault**, which remains in existence indefinitely if no corrective action is taken. Second, there is the **transient fault**, which can appear and disappear within a very short period of time. Third, there is the **intermittent fault**, which appears, disappears, and then reappears repeatedly. An example of a permanent fault is a logic line that is physically stuck at a logic 1. An example of a transient fault is one resulting from some external disturbance such as lightning. The lightning can temporarily disrupt a system but do no permanent damage; as soon as the lightning disappears, the fault can also disappear. The effects of the external disturbance, that is, the errors, can take some time to disappear, but the actual fault may be present for only a very short time. An example of an intermittent fault is one resulting from a weak solder joint in a circuit. The solder joint can provide proper contact at certain times and improper contact at others.

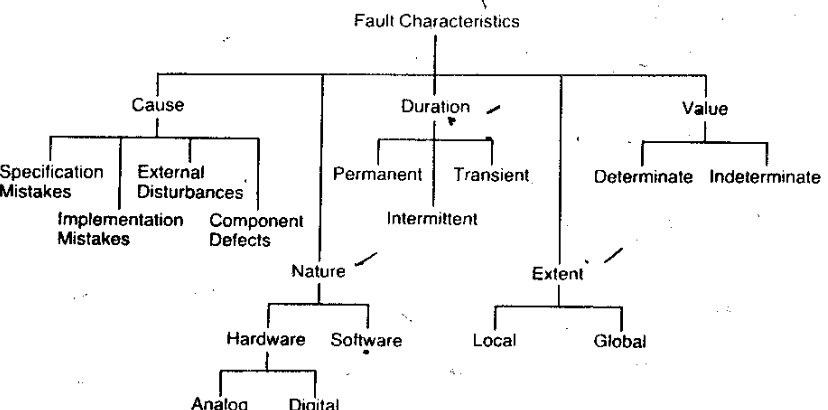


Fig. 2.7 Faults can be characterized by five attributes: cause, nature, duration, extent, and value.

The **fault extent** specifies whether the fault is localized to a given hardware or software module or whether it globally affects the hardware, the software, or both. A power supply fault is a good example of a fault that might globally affect a system. On the other hand, a fault in a memory that is used by only one processor might have a very localized impact on a system.

The **fault value** can be either determinate or indeterminate. A **determinate fault** is one whose status remains unchanged throughout time unless externally acted upon. For example, a fault that always results in a line being a logical 1 is a determinate fault. An **indeterminate fault** is one whose status at some time T may be different from its status at some increment of time greater than or less than T. For example, a number of hardware faults can produce state oscillations between a logical 1 and a logical 0. A good example is a fault that is sensitive to either the data or time.

2.5 Fault Models

In much of our work, it is necessary to assume that faults behave according to some fault model [Hayes 1985]. Although in practice faults can be transient in duration and indeterminate in value, it can be extremely difficult to analyze digital systems if we assume that faults have these characteristics. This is particularly true when we attempt to design test procedures for digital systems or to simulate faults within such systems. To make our problems more manageable, we need some way to restrict our attention to a subset of all faults that can occur.

Fault models allow us to specifically define the types of faults that will be considered and the behavior these faults will have. In addition, fault models allow us to represent the behavior of physical occurrences. We use models in all disciplines of engineering, and although we understand that models are not 100% accurate in all cases, we use them to make problems tractable and because their usage often introduces little error. The same is true of fault modeling.

Fault models attempt to represent the types of faults that can occur. In this section, we consider two primary fault models: the logical stuck-fault model that is used at the logic circuit level and the transistor stuck-fault model that is used at the transistor circuit level.

2.5.1 The Logical Stuck-Fault Model

The most common fault model is the **logical stuck-fault model** [Kohavi 1978], which has gained its popularity because of its effectiveness and simplicity. The logical stuck-fault model is sometimes referred to as the stuck-at-0 (s-a-0), stuck-at-1 (s-a-1) fault model or simply the stuck-fault model.

There are three basic assumptions of the stuck-fault model:

- a fault results in a module responding as if one of its inputs or outputs is physically stuck at a logic 1 or 0
- the basic functionality of the circuit is not altered by the fault
- the fault is permanent

The logic module can be a single gate or a collection of gates that implements some logic function.

The first assumption of the stuck-fault model is further illustrated in Fig. 2.8 by an AND gate having inputs A and B and output F. The input line A is assumed to remain free to take on either logic value (1 or 0), but the gate responds as if the line is physically stuck at either a logic 1 or logic 0. For example, a stuck-at-1 fault on line A results in the output of the AND gate in Fig. 2.8 being 1 whenever input B is 1, regardless of the actual value applied to line A. Line A is free, however, to assume any logic value. Likewise, a stuck-at-0 fault on line A causes the AND gate to always have an output of 0, regardless of the actual value applied to input A.

The second assumption of the stuck-fault model is that the basic functionality of the circuit is not affected by the fault. This assumption is often confusing, but it is very important in the stuck-fault model. It does not mean that the circuit continues to produce the correct results, but instead simply implies that the circuit produces the results expected of it, given the existence of the fault. For example, an AND gate that has a stuck-at-0 fault on one of its inputs should always produce a 0 at the output if the gate continues to behave as an AND gate. If, however, the fault transforms the AND

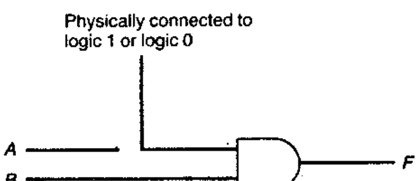


Fig. 2.8 Illustration of the basic concept of the logical stuck-fault model. The gate responds as if the input is permanently connected to a 1 or 0. The applied input, however, is free to assume any value.

gate into an EXCLUSIVE-OR gate, the assumptions of the stuck-fault model are violated. Likewise, if a fault transforms a combinational circuit into a sequential circuit, the assumptions of the stuck-fault model are violated. This latter example is a significant problem in many of today's integrated circuit technologies.

The third assumption of the stuck-fault model is that the fault is permanent. Simply stated, the faulty module *always* performs as if a line is stuck at a specific logic value. This assumption simplifies the fault model by avoiding the difficulty of modeling intermittent or transient faults.

Most applications of the stuck-fault model restrict the number of faults that can occur at any one time. Typically, we assume that a circuit will never have more than one stuck fault. Although the single-fault assumption is not a specific property of the stuck-fault model, it is a very common assumption that is used in conjunction with the remaining stuck-fault assumptions to simplify the process of analyzing a circuit or generating test patterns. In a circuit that contains n lines, at most $2n$ unique, single, stuck faults can occur. Consequently, an upper bound is placed on the number of faults that must be examined. Even with the bound, it is easy to see that large circuits can have a prohibitively large number of potential faults.

The effectiveness of a fault model can often be quantified by a coverage parameter. A fault model is said to cover a fault if and only if the actual physical fault is accurately represented by the chosen fault model. Ideally, one hopes that a fault model covers 100% of all physical faults, but this is seldom the case.

The classic example of a case where the stuck-fault model does not cover a very specific and practical physical fault is found in the Complementary Metal Oxide Semiconductor (CMOS) implementation of a two-input NOR gate [Wadsack 1978]. The logic diagram and the transistor realization of the CMOS NOR gate are shown in Fig. 2.9. The circuit is a combination of two p-channel transistors in series and two n-channel transistors in parallel. Based on the values of the inputs, A and B, a path for current flow is established from either V_{DD} or V_{SS} to the output of the circuit. For example, if both A and B are at the logic 0 level, both p-channel transistors are conducting while both n-channel transistors are turned off. A path is established between V_{DD} and the output, thereby forcing the output to a logic 1 value. In a similar manner, if either or both inputs are 1, the corresponding p-channel transistors are turned off while one or both n-channel transistors are turned on; the result is that a path is established from the output to V_{SS}, thus forcing the output to a logic 0 value.

Several faults that behave as stuck faults can be easily recognized in the NOR circuit. For example, if input line A becomes physically shorted to

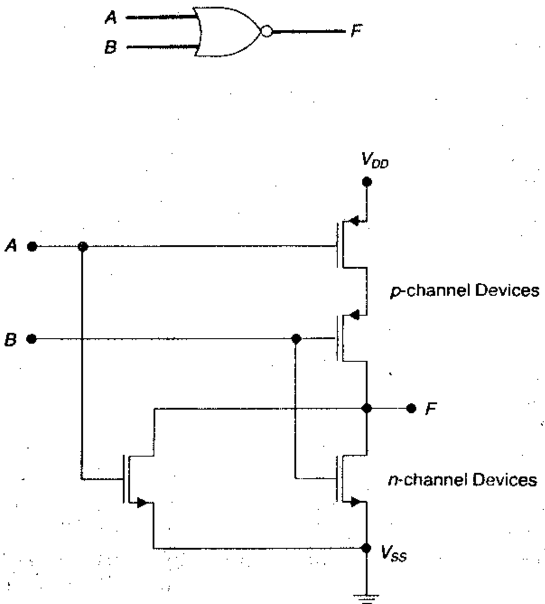


Fig. 2.9 Logic diagram and transistor implementation of the CMOS NOR gate. Inputs A and B determine if the output is pulled high (V_{DD}) or low (V_{SS}).

V_{DD}, the circuit behaves as if line A is stuck at 1; specifically, the output of the circuit will always be 0. Another example is when the drain and source of one of the n-channel devices become shorted together, causing the device to always have a logic 0 on the output. Consequently, the fault can be modeled as the output stuck at 0.

Several faults do not adhere to the stuck-fault model. One example is when a line within the circuit breaks. If a break in a line occurs, as shown in Fig. 2.10, and the input combination $AB = 10$ is presented to the circuit, a path does not exist from either V_{DD} or V_{SS} to the circuit output. In other words, neither the series network of p-channel transistors nor the parallel network of n-channel transistors is conducting. The output under such conditions will be pulled neither to logic 0 nor to logic 1, but, due to load capacitances present on the output, the output retains the value defined by the previous input. The length of time that the output remains at the previous value depends on the length of time required to discharge the load capacitance. This type of fault is often referred to as a **stuck-open fault.**

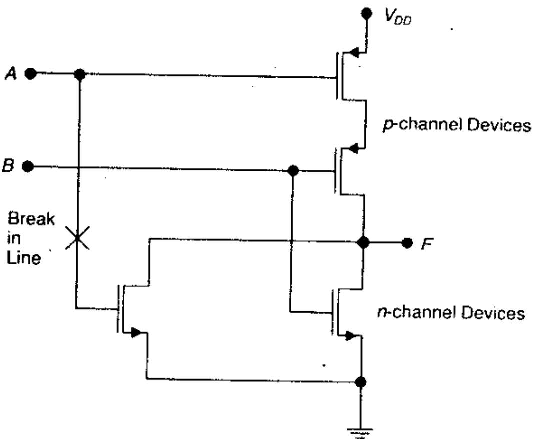


Fig. 2.10 A stuck-open fault exists when a break in a line occurs. Certain input patterns result in the output not being pulled either high or low. Instead the load capacitances on the output result in a previously defined output being retained.

When a fault results in the present output value depending on the previous output, the circuit has obtained a form of memory and is, therefore, no longer a combinational circuit. Instead, the circuit is a sequential circuit, and one of the basic assumptions of the stuck-fault model is violated. Therefore, the stuck-fault model cannot adequately model the stuck-open fault that can occur in the CMOS NOR gate.

Recognizing the limitations of the stuck-fault model, researchers have attempted to develop new and better fault models. For example, in [Wadsack 1978] logic circuit models of the various gates, such as the NOR gate, are developed to allow the effect of the stuck-open fault in CMOS circuits to be simulated. However, the fault model requires that additional gates be added to the description of the circuit, thereby increasing the complexity.

As an example, Fig. 2.11 shows the modified representation of a CMOS NOR gate. In Fig. 2.11, the D-type flip flop is used to model the memory that is introduced when the stuck-open fault occurs. The flip flop is a level-triggered device such that when the clock line is 1, the output of the flip flop simply follows the D input. When the clock transitions from a 1 to a 0, however, the value on the D input is latched, and the flip flop output retains that value until the clock returns to 1. Note that under all fault-free operating conditions, the clock remains at a logic 1, and the output of the flip flop is just the output of the NOR gate G_1. Traditional stuck-at-1 and stuck-at-0

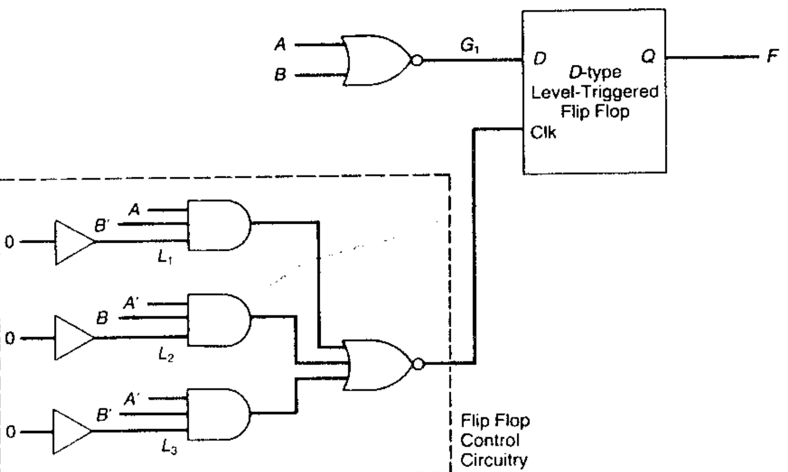


Fig. 2.11 Logical model of a NOR gate to account for the stuck-open fault. The D-type flip flop is used to simulate the memory effect produced by the stuck-open fault.

faults can be injected by simply placing them on the inputs A and B, and on the output G_1 of the NOR gate.

Stuck-open faults can be modeled by placing a stuck-at-1 fault on either line L_1, L_2, or L_3. L_1 stuck-at-1 represents a stuck-open fault affecting the n channel device driven by input A. L_2 stuck-at-1 represents a stuck-open fault affecting the n-channel device driven by input B. Finally, L_3 stuck-at-1 represents a stuck-open fault affecting either p-channel device. For example, if L_1 is stuck-at-1, the clock line on the flip flop will be logic 0 whenever the input combination $AB = 10$ occurs. In other words, the n-channel device driven by input A should be on, but it is not because of the stuck-open fault. Consequently, the output of the gate is simply the output specified by the previous input combination. Any other input combination results in the gate performing as expected.

The difficulty with the modeling scheme developed in [Wadsack 1978] is the additional complexity that must be added to a circuit to allow for the modeling of stuck-open faults. For each gate, we must add four additional gates and a flip flop to allow the effects of stuck-open faults to be adequately modeled. Consequently, the complexity of the circuit, as far as simulation is concerned, is substantially increased.

2.5.2 Transistor Stuck-Fault Models

In some cases, such as the stuck-open fault, the logical stuck-fault model is inaccurate. One way to make the fault model more accurate is to construct the model at the transistor level as opposed to the gate level. Such models are often referred to as switch-level models because the transistor is essentially performing a switching function. Here, however, we use the terminology **transistor stuck-fault model** to describe the techniques employed.

In the transistor stuck-fault model, faults are represented as either a transistor stuck on or a transistor stuck off. For example, in the CMOS NOR gate of Fig. 2.9, the faults are assumed to result in transistors being permanently on or permanently off. The advantage of the transistor stuck-fault model is that the stuck-open fault described previously can be represented as a transistor permanently stuck off. For example, in Fig. 2.9, if the n-channel transistor driven by input A is permanently off, the response of the circuit to the input combination $AB = 10$ will be that the output is forced neither to V_{DD} nor to V_{SS}. Consequently, the output retains its previous value due to the load capacitances present on the output.

The clear disadvantage of the transistor stuck-fault model is the additional complexity. For example, a circuit containing 100 NOR gates contains 400 transistors, so the number of elements that must be represented and manipulated has quadrupled. Other gates such as EXCLUSIVE-ORs require even more transistors. In circuits containing many gates, the impact on complexity of using the transistor stuck-fault model can be overwhelming.

2.6 Error Models

Another technique that has been proposed for the modeling of the effects of faults is to model the effect in the informational universe. In other words, the *error* is modeled rather than the fault [Patel and Fung 1982]. The underlying assumption of this approach is that regardless of the fault, the effect of that fault in the informational universe is to change a logic value at some time and in some result produced by the system. For combinational circuits, this can be viewed as a modification of the circuit's truth table.

The truth table modification that results can vary as a function of time, but the truth table will be either correct or modified in some way. In other words, we may not know what form the fault takes, but we can look at the response of the circuit and determine whether or not the results are correct. We use this type of model in our daily lives when we compare pieces of information. For example, when we balance our checkbooks, we detect the existence of an error by the deviation between the bank's balance and our balance. Once we have detected the error, we must then examine the details

of the checkbook and the statement to locate the problem. However, if the bank's balance agrees with our own, we can be reasonably confident that both balances are correct.

Error models are useful in the design and verification of many self-testing schemes, but they are not normally used as a means of test pattern generation, as the stuck-fault model is. Also, the error model implies that some time can elapse before we recognize that a fault exists.

Various other forms of fault models have been developed in the past for modeling faults in digital systems [Hayes 1985]. For the purposes of this text, however, the logical stuck-fault model is used during our discussions of test pattern generation. The interested reader is referred to the additional reading for descriptions of other available fault models.

2.7 Design Philosophies to Combat Faults

There are three primary techniques for attempting to improve or maintain a system's normal performance in an environment where faults are of concern: fault avoidance, fault masking, and fault tolerance. Figure 2.12 illustrates the barriers that are constructed by each of the available techniques.

Fault avoidance is any technique that is used to prevent faults in the first place. Fault avoidance can include such things as design reviews, component screening, testing, and other quality control methods. If a design review, for example, is conducted appropriately, many of the specification mistakes that might otherwise result in faults can be eliminated. Also, a system can often be shielded to prevent external disturbances such as lightning or radiation from causing faults in the system. Shielding is a form of fault avoidance. Finally, if a design is effectively tested, many of the faults that might be in a system after manufacture can be detected and eliminated before the system is placed into operation.

Fault masking is any process that prevents faults in a system from introducing errors into the informational structure of that system. Error correcting memories, for example, correct a memory's data before a system uses the data. Thus the system never experiences the impact of the fault within the memory. Error correction in a memory is a form of fault masking. Another example of fault masking is majority voting. If a committee of three people makes a decision by voting yes or no on a proposition, any two votes that agree determine the decision of the committee. The ultimate decision of the committee represents the wishes of a majority of the committee members and masks the desires of a member that happens to disagree with the majority. Similar techniques can be applied to digital systems such that two modules can mask the effect of a faulty module.

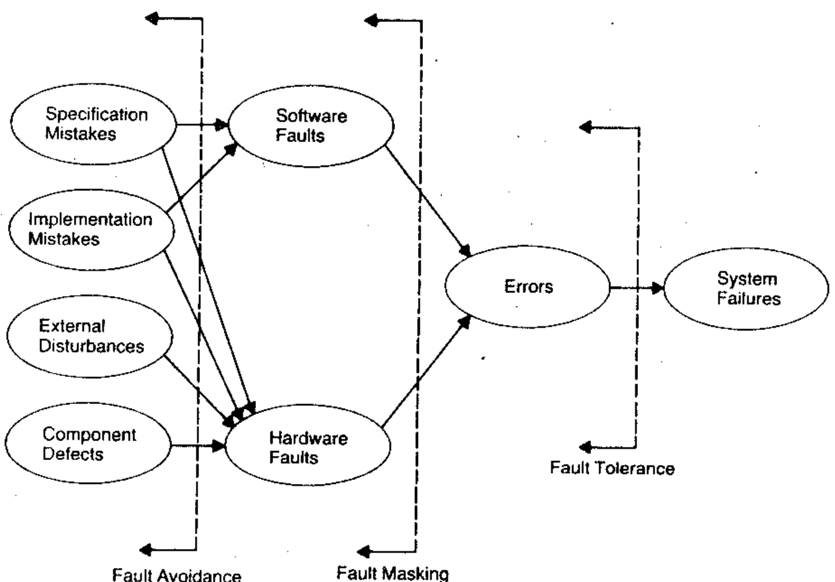


Fig. 2.12 Barriers constructed by design techniques of fault avoidance, fault tolerance, and fault masking.

Fault tolerance is the ability of a system to continue to perform its tasks after the occurrence of faults. The ultimate goal of fault tolerance is to prevent system failures from ever occurring. Since failures are directly caused by errors, the terms fault tolerance and error tolerance are often used interchangeably. For our purposes, however, we use the term fault tolerance.

Fault tolerance can be achieved by many techniques. Certainly, fault masking is one approach to tolerating faults that have occurred. Another approach is to detect and locate the fault that has occurred and reconfigure the system to remove the faulty component. **Reconfiguration** is the process of eliminating a faulty entity from a system and restoring the system to some operational condition or state. If the reconfiguration technique is used, the designer must be concerned with the following processes:

1. **Fault detection** is the process of recognizing that a fault has occurred. Fault detection is often required before any recovery procedure can be implemented.
2. **Fault location** is the process of determining where a fault has occurred so that an appropriate recovery can be implemented.

3. **Fault containment** is the process of isolating a fault and preventing the effects of that fault from propagating throughout a system. Fault containment is required in all fault-tolerant designs.
4. **Fault recovery** is the process of remaining operational or regaining operational status via reconfiguration even in the presence of faults.

Equivalent definitions can be provided in the informational universe:

1. **Error detection** is the process of recognizing that an error has occurred.
2. **Error location** is the process of determining which specific module produced the error.
3. **Error containment** is the process of preventing the error from propagating throughout a system.
4. **Error recovery** is the process of regaining operational status or restoring the system's integrity after the occurrence of an error.

A typical fault-tolerant system that employs reconfiguration performs as follows. Suppose a fault occurs in the system. Fault detection techniques can identify that a fault exists, and fault location procedures specifically identify the source of the fault. Fault recovery and reconfiguration techniques then remove the faulty module and either replace it with a fault-free module or degrade the functionality of the system to keep it operational.

Summary

This chapter has presented the definitions of several terms that are crucial to an understanding of fault-tolerant system design. In addition, we have discussed the basic causes and characteristics of faults in digital systems. Each fundamental term and concept is summarized in the following list.

Component Defects—physical imperfections or flaws in an electronic component.

Determinate Fault—a fault whose status remains unchanged throughout time.

Error—the occurrence of an incorrect value in some unit of information within a system.

Error Containment—the process of preventing an error from propagating throughout a system.

Error Detection—the process of recognizing that an error has occurred.

Error Latency—the length of time between the occurrence of an error and the appearance of a system failure.

Error Location—the process of determining the source of an error.

Error Recovery—the process of regaining operational status and restoring a system's integrity after the occurrence of an error.

External Disturbances—an action external to the system that produces a hardware or software fault.

External Universe—the domain where the user of a system ultimately sees the effects of faults and errors. The external universe is where failures occur.

Failure—a deviation in the expected performance of a system.

Fault—a physical defect, imperfection, or flaw that occurs in hardware or software.

Fault Avoidance—a technique that attempts to prevent the occurrence of faults.

Fault Cause—one of four basic items that can result in faults: specification mistakes, implementation mistakes, component defects, and external disturbances.

Fault Containment—the process of confining the effects of a fault to a limited locality.

Fault Detection—the process of recognizing that a fault has occurred.

Fault Duration—the length of time that a fault is active in a system.

Fault Extent—the characteristic that specifies whether a fault is localized to a given module or globally affects the system.

Fault Latency—the length of time between the occurrence of a fault and the appearance of an error.

Fault Location—the process of determining where a fault has occurred.

Fault Masking—the process of preventing faults from introducing errors.

Fault Nature—a characteristic of a fault that describes its type. The fault type can be hardware, software, digital, or analog.

Fault Recovery—the process of maintaining or regaining operational status after a fault has occurred.

Fault Tolerance—the ability to continue the correct performance of functions in the presence of faults.

Fault Value—the characteristic that specifies whether a fault is determinate or indeterminate.

Implementation Mistakes—incorrect actions occurring during the transformation of hardware and software specifications into physical hardware and actual software.

Indeterminate Fault—a fault whose status changes from time to time.

Informational Universe—the domain that contains units of information and is the location of errors.

Intermittent Fault—a fault that appears, disappears, and then reappears within a system.

Latent Fault—a fault that is present within a system but that has not yet produced an error.

Logical Stuck-Fault Model—a representation that assumes all faults will appear as lines in the logic diagram being physically stuck at a logic 1 or logic 0 value.

Malfunction—the incorrect performance of some system function. A malfunction is equivalent to a failure.

Permanent Fault—a fault that remains in a system indefinitely.

Physical Universe—a domain that contains physical entities and is the location of faults.

Reconfiguration—the process of eliminating a faulty entity from a system and restoring the system to some operational state.

Specification Mistakes—incorrect algorithms, architectures, or hardware/software design specifications that lead to either hardware or software faults.

Stuck-open Fault—a physical fault resulting in the output of a gate depending on the present input and the previous output.

Transient Fault—a fault that appears and then disappears a short time later.

Transistor Stuck-Fault Model—a representation that assumes all faults will appear as transistors in the circuit diagram being physically stuck on or stuck off.

User's Universe—equivalent to the external universe. The domain where the user sees the effects of faults and errors.

References

1. Avizienis, A. "The four-universe information system model for the study of fault tolerance," *Proceedings of the 12th Annual International Symposium on Fault-Tolerant Computing*, Santa Monica, California, June 22-24, 1982, pp. 6–13.
2. Hayes, J.P. "Fault modeling," *IEEE Design and Test*, Vol. 2, No. 2, April 1985, pp. 88–95.
3. Johnson, B.W. "Fault-tolerant microprocessor-based systems," *IEEE Micro*, Vol. 4, No. 6, December 1984, pp. 6–21.
4. Kohavi, Z. *Switching and Finite Automata Theory*, McGraw-Hill, New York, 1978.
5. Laprie, J-C. "Dependable computing and fault tolerance: Concepts and terminology," *Proceedings of the 15th Annual International Symposium on Fault-Tolerant Computing*, Ann Arbor, Mich., June 19-21, 1985, pp. 2–11.
6. Nelson, V.P., and B.D. Carroll. "Fault-tolerant computing (A tutorial)," presented at the AIAA Fault Tolerant Computing Workshop, November 8-10, 1982, Fort Worth, Tex.
7. Patel, J.H., and L.Y. Fung. "Concurrent error detection in ALUs by recomputing with shifted operands," *IEEE Transactions on Computers*, Vol. C-31, No. 7, July 1982, pp. 589–595.
8. Wadsack, R.L. "Fault modeling and logic simulation of CMOS and MOS integrated circuits," *The Bell System Technical Journal*, Vol. 57, No. 5, May-June 1978, pp. 1449–1475.

Additional Reading

The following list of references is provided for the reader who is interested in pursuing the topics of this chapter in more detail.

Anderson, T., and P.A. Lee. *Fault Tolerance Principles and Practices*, Prentice-Hall International, London, 1981.

Anderson, T., and P.A. Lee. "Fault tolerance terminology proposals," *Proceedings of the 12th Annual International Symposium on Fault-Tolerant Computing*, Santa Monica, Calif., June 22-24, 1982, pp. 29–33.

Avizienis, A. "Fault tolerance: The survival attribute of digital systems," *Proceedings of the IEEE*, Vol. 66, No. 10, October 1978, pp. 1109–1125.

Banerjee, P., and J.A. Abraham. "Characterization and testing of physical failures in MOS logic circuits," *IEEE Design and Test*, Vol. 1, No. 4, August 1984, pp. 76–86.

Beh, C.C., K.H. Arya, C.E. Radke, and K.E. Torku. "Do stuck fault models reflect manufacturing defects?" *Proceedings of the 1982 International Test Conference*, October 1982, pp. 35–42.

Bryant, R.E. "A switch-level model and simulator for MOS digital systems," *IEEE Transactions on Computers*, Vol. C-33, No. 2, February 1984, pp. 160–177.

Bryant, R.E., and M.D. Schuster. "Fault simulation of MOS digital circuits," *VLSI Design*, Vol. 4, No. 10, October 1983, pp. 24–30.

Carter, W.C. "A time for reflection," *Proceedings of the 12th Annual International Symposium on Fault-Tolerant Computing*, Santa Monica, Calif., June 22–24, 1982, p. 41.

Case, G.R. "Analysis of actual fault mechanisms in CMOS logic gates," *Proceedings of the 13th Design Automation Conference*, June 1976, pp. 265–270.

Chandramouli, R. "On testing stuck-open faults," *Proceedings of the Fault Tolerant Computing Symposium*, 1983, pp. 258–265.

Courtois, B. "Failure mechanisms, fault hypothesis, and analytical testing on LSI NMOS (HMOS) circuits," *VLSI 81*, University of Edinburgh, August 1981, Academic Press, 1981.

El-ziq, Y.M. "Classifying, testing, and eliminating VLSI MOS failures," *VLSI Design*, Vol. 4, No. 9, September 1983, pp. 30–35.

El-ziq, Y.M. "Automatic test generation for stuck-open faults in CMOS VLSI," *Proceedings of the 18th Design Automation Conference*, June 1981, pp. 347–352.

Galiay, J., Y. Crouzet, and M. Verniault. "Physical versus logical fault models in MOS LSI circuits: Impact on their testability," *IEEE Transactions on Computers*, Vol. C-29, No. 6, June 1980, pp. 527–531.

Goldberg, J. "A time for integration," *Proceedings of the 12th Annual International Symposium on Fault-Tolerant Computing*, Santa Monica, Calif., June 22–24, 1982, p. 42.

Gupta, A.K., and J.R. Armstrong. "Functional fault modeling and simulation for VLSI devices", Department of Electrical Engineering, Virginia Polytechnic Institute and State University, Blacksburg, Va., 1985.

Hayes, J.P. "Modeling faults in digital logic circuits," in *Rational Fault Analysis*, R. Saeks and S.R. Liberty, ed., Marcel-Dekker, New York, 1977, pp. 78–95.

Hayes, J.P. "Fault modeling for digital MOS integrated circuits," *IEEE Transactions on Computer-Aided Design*, Vol. CAD-3, No. 3, July 1984, pp. 200–207.

Hayes, J.P. "An introduction to switch-level modeling," *IEEE Design and Test of Computers*, Vol. 4, No. 4, August 1987, pp. 18–25.

Kopetz, H. "The failure-fault (FF) model," *Proceedings of the 12th Annual International Symposium on Fault-Tolerant Computing*, Santa Monica, California, June 22-24, 1982, pp. 14–17.

Lala, P.K. *Fault Tolerant and Fault Testable Hardware Design*, Prentice-Hall International, London, England, 1985.

Lee, P.A., and D.E. Morgan. "Fundamental concepts of fault-tolerant computing—progress report," *Proceedings of the 12th Annual International Symposium on Fault-Tolerant Computing*, Santa Monica, California, June 22-24, 1982, pp. 34–38.

Mangir, T.E. "Sources of failure and yield improvement for VLSI," *Proceedings of the IEEE*, Vol. 72, No. 6, June 1984, pp. 690–708.

Pradhan, D.K. *Fault-Tolerant Computing—Theory and Techniques*, Volumes I and II, Prentice-Hall, Englewood Cliffs, N.J., 1986.

Reddy, S.M., M.K. Reddy, and V.D. Agrawal. "Robust tests for stuck-open faults in CMOS combinational logic circuits," *Proceedings of the 14th International Fault Tolerant Computing Symposium*, June 1984, pp. 44–49.

Rennels, D.A. "Fault-tolerant computing—concepts and examples," *IEEE Transactions on Computers*, Vol. C-33, No. 12, December 1984, pp. 1116–1129.

Robinson, A.S. "A user oriented perspective of fault-tolerant systems models and terminologies," *Proceedings of the 12th Annual International Symposium on Fault-Tolerant Computing*, Santa Monica, Calif., June 22-24, 1982, pp. 22–28.

Siewiorek, D.P., and R.S. Swarz. *The Theory and Practice of Reliable System Design*, Digital Press, Bedford, Mass., 1982.

Timoc, C., M. Buehler, T. Griswold, C. Pina, F. Stott, and L. Hess. "Logical models of physical failures," *Proceedings of the International Test Conference*, 1983, pp. 546–553.

Problems

2.1. Devise an original example to illustrate the differences between faults, errors, and failures. As you illustrate these concepts, relate them to the three-universe model.

2.2. Some systems are designed for reliability, whereas others are designed to be highly available. Explain the difference between these two concepts and give an original example of an application requiring high reliability and one that needs high availability. How would you expect the four different causes of faults to complicate the designs of high reliability applications as compared to high availability applications?

2.3. Faults can be characterized by five major attributes. Give original examples of faults that illustrate each of these attributes.

2.4. Develop a design methodology for digital systems that incorporates fault avoidance at each of the major steps. In other words, examine the design process and determine where fault avoidance techniques such as designs reviews should be incorporated. Pretend you are making recommendations to your employer on how fault avoidance can be used to improve the quality of your company's products.

2.5. Fault masking is an attractive technique for use in systems that cannot allow even momentary erroneous results to be generated. However, fault masking does have several serious limitations. In your opinion, what are the disadvantages of using a fault masking approach?

2.6. Develop a glossary of the key terms used in the fault-tolerant computing field. Keep the glossary available to use as a reference and study guide.

2.7. Explain the difference between the stuck-at-1, stuck-at-0 fault model and the transistor stuck-fault model. Use the transistor diagram of a CMOS NAND gate to illustrate a physical fault that is covered by the stuck-at-1, stuck-at-0 fault model but is not covered by the transistor stuck-fault model. Also, determine a physical fault that is covered by the transistor stuck-fault model but is not covered by the stuck-at-1, stuck-at-0 fault model.

2.8. The stuck-at-0, stuck-at-1 fault model was developed when transistor transistor logic (TTL) was extremely popular. Contrast the differences between TTL and CMOS, and develop an explanation of whether the stuck-at-0, stuck-at-1 fault model is more appropriate for TTL than CMOS. If the model is more appropriate, provide a detailed discussion of why.

4

Evaluation Techniques

4.1 Introduction
4.2 Quantitative Evaluation Methods
4.3 Reliability Modeling
4.4 Safety Modeling
4.5 System Comparisons
4.6 Availability Models
4.7 Maintainability Models
4.8 Redundancy Ratios
4.9 Qualitative Methods
4.10 Tradeoff Analysis Example
Summary
References
Additional Reading
Problems

4.1 Introduction

The techniques presented in the previous chapter form a collection of approaches that can be used to achieve fault tolerance. The specific techniques used in a given application depend on not only the application, but also on the ideas and philosophies of the designers. One team of designers might develop one fault-tolerant design, whereas another team would choose a completely different approach for the same application. It is important to be

able to compare two or more approaches for a particular application. The process of comparison is actually a critical part of the design process because it leads to tradeoffs and modifications of the design. It is through such tradeoffs that the most optimal design is developed.

The methods for evaluating fault-tolerant systems can be divided into two major categories: *quantitative* and *qualitative*. Qualitative measures are typically subjective in nature and describe the benefits of one design over another. Examples include the flexibility of a particular design and the degree to which the fault tolerance techniques are transparent to the user of a system. Quantitative evaluation techniques produce numbers that can be used to compare two or more systems. Examples include specific numbers for the reliability, availability, maintainability, mission life, or fault coverage of a system.

The purpose of this chapter is to examine, in detail, the techniques that are available for evaluating fault-tolerant systems. The final section contains a specific tradeoff example that shows how the evaluation techniques can be used to make design decisions and tradeoffs.

4.2 Quantitative Evaluation Methods

As previously mentioned, the purpose of quantitative evaluation methods is to assign a number to some attribute of a system such that the attribute can be compared among systems. For example, the reliability of one system may be greater than that of another, or the weight and cost of one approach may be less than that of another. Reliability, weight, and cost are three examples of quantitative measures that can be used to evaluate systems. Certainly, the weight and cost of a system are extremely important, and in some cases can be more important than reliability or fault tolerance.

In this chapter, we consider several approaches to quantitative evaluation, including the failure rate, mean time to failure (MTTF), mean time between failure (MTBF), fault coverage, reliability analysis, safety analysis, availability analysis, maintainability analysis, redundancy ratios, and costs. Several techniques for generating the reliability of a system are presented.

4.2.1 Failure Rate and the Reliability Function

Intuitively, the **failure rate** is the expected number of failures of a type of device or system per a given time period [Shooman 1968]. For example, if a computer fails, on the average, once every 2000 hours, the computer has a failure rate of one failure per 2000 hours, or 1/2000 failures/hour. The failure rate is typically denoted as λ. The failure rate is one measure that can be used to compare systems or components. In selecting a computer for a

banking application, one would like to select a computer that fails as infrequently as possible. If redundancy has been incorporated as a means of achieving fault tolerance, the failure rate of the redundant system should be lower than the failure rate of a similar, nonredundant system.

To more clearly understand the mathematical basis for the concept of a failure rate, recall the definition of the reliability function. The reliability $R(t)$ of a component, or a system, is the conditional probability that the component operates correctly throughout the interval $[t_0,t]$ given that it was operating correctly at time t_0. Suppose that we test N identical components by placing all N components in operation at time t_0 and recording the number of failed and working components at time t. Let $N_f(t)$ be the number of components that have failed at time t and $N_o(t)$ be the number of components that are operating correctly at time t. It is assumed that once a component fails it remains failed indefinitely. The *reliability* of the components at time t is given by

$$R(t) = \frac{N_o(t)}{N} = \frac{N_o(t)}{N_o(t) + N_f(t)}$$

which is simply the probability that a component has survived the interval $[t_o,t]$. The probability that a component has not survived the time interval is called the *unreliability* and is given by

$$Q(t) = \frac{N_f(t)}{N} = \frac{N_f(t)}{N_o(t) + N_f(t)}$$

Note that at any time t, $R(t)=1.0 - Q(t)$ because

$$R(t) + Q(t) = \frac{N_o(t) + N_f(t)}{N_o(t) + N_f(t)} = 1.0$$

If we write the reliability function as

$$R(t) = 1.0 - \frac{N_f(t)}{N}$$

and differentiate $R(t)$ with respect to time, we obtain

$$\frac{dR(t)}{dt} = -\frac{1}{N}\frac{dN_f(t)}{dt}$$

which can be written as

$$\frac{dN_f(t)}{dt} = -N\frac{dR(t)}{dt}$$

The derivative of $N_f(t)$, $dN_f(t)/dt$, is simply the instantaneous rate at which components are failing. At time t, there are still $N_o(t)$ components operational. Dividing $dN_f(t)/dt$ by $N_o(t)$ we obtain

$$z(t) = \frac{1}{N_o(t)} \frac{dN_f(t)}{dt}$$

$z(t)$ is called the ***hazard function***, ***hazard rate***, or ***failure rate function***. The units for the failure rate function are failures per unit of time.

The failure rate function can be expressed in different ways. For example, $z(t)$ can be written strictly in terms of the reliability function $R(t)$ as

$$z(t) = \frac{1}{N_o(t)} \frac{dN_f(t)}{dt} = \frac{1}{N_o(t)} \left[-N \frac{dR(t)}{dt} \right] = -\frac{\frac{dR(t)}{dt}}{R(t)}$$

Similarly, $z(t)$ can be written in terms of the unreliability $Q(t)$ as

$$z(t) = -\frac{\frac{dR(t)}{dt}}{R(t)} = \frac{\frac{dQ(t)}{dt}}{1 - Q(t)}$$

The derivative of the unreliability $dQ(t)/dt$ is called the *failure density function*.

The failure rate function clearly depends on time because the value of $N_o(t)$ and the value of $dN_f(t)/dt$ change as functions of time. However, experience has shown that the failure rate function for electronic components does have a period where the value of $z(t)$ is approximately constant. The commonly accepted relationship between the failure rate function and time for electronic components is called the **bathtub curve** and is illustrated in Fig. 4.1. The bathtub curve assumes that during the early life of systems failures occur frequently due to substandard or weak components. The decreasing part of the bathtub curve is called the *early-life* or *infant mortality* region. At the opposite end of the curve is the wear-out region where systems have been functional for a long period of time and are beginning to experience failures due to the physical wearing of electronic or mechanical components. The increasing part of the bathtub curve is called the *wear-out phase*. During the intermediate region, the failure rate function is assumed to be a constant. The constant portion of the bathtub curve is called the *useful life* phase of the system, and the failure rate function is assumed to have a value of λ during that period. (λ is referred to as the failure rate and is normally expressed in units of failures per hour.)

The period of a constant failure rate is typically the most useful portion of a system's life. During the useful-life phase, the system is providing its most predictable service to its users. We usually attempt to get a system beyond the infant mortality stage by using the concept of *burn-in* to remove weak components. Burn-in implies operating a system, often at an acceler-

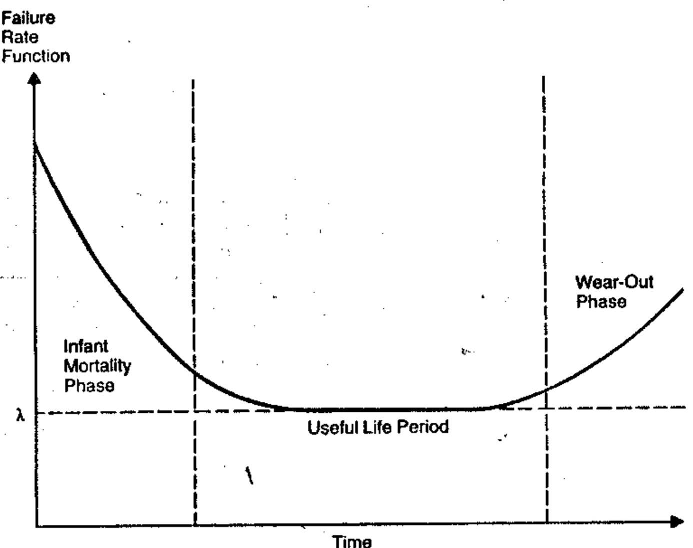


Fig. 4.1 Bathtub curve relationship between the failure rate function and time—the failure rate function is constant during the useful life period.

ated pace, prior to placing the system into service to get the system to the beginning of the useful-life period. In addition, the system is normally replaced before it enters the wear-out phase of its life. Thus, the primary interest is the performance of the system during the useful-life phase.

As noted previously, the failure rate function can be related to the reliability function as

$$z(t) = \frac{1}{N_o(t)} \frac{dN_f(t)}{dt} = -\frac{N}{N_o(t)} \frac{dR(t)}{dt}$$

We know, however, that the quantity $N/N_o(t)$ is the inverse of the reliability function $R(t)$ so we can write

$$z(t) = -\frac{1}{R(t)} \frac{dR(t)}{dt}$$

The result is a differential equation of the form

$$\frac{dR(t)}{dt} = -z(t)R(t)$$

If we assume that the system is in the useful-life stage where the failure rate function has a constant value of λ, the solution to the differential equation is well known to be an exponential function of the parameter λ given by

$$R(t) = e^{-\lambda t}$$

where λ is the constant failure rate. The exponential relationship between the reliability and time is known as the **exponential failure law**, which states that for a constant failure rate function, the reliability varies exponentially as a function of time.

The exponential failure law is extremely valuable for the analysis of electronic components and is by far the most commonly used relationship between reliability and time. Many cases, however, cannot assume that the failure rate function is constant, so the exponential failure law cannot be used; other modeling schemes and representations must be employed. An example of a time-varying failure rate function is found in the analysis of software. Software failures are the result of design faults, and as a software package is used, design faults are discovered and corrected. Consequently, the reliability of software should improve as a function of time, and the failure rate function should decrease.

A common modeling technique used to represent time-varying failure rate functions is the **Weibull distribution** [Siewiorek and Swarz 1982]. The failure rate function associated with the Weibull distribution is given by

$$z(t) = \alpha\lambda(\lambda t)^{\alpha-1}$$

where α and λ are constants that control the variation of the failure rate function with time. The failure rate function given by the Weibull distribution is intuitively appealing. For example, if the value of α is 1, $z(t)$ is simply the constant λ. If α is greater than 1, $z(t)$ increases as time increases; if α is less than 1, $z(t)$ decreases as time increases. Consequently, we can envision modeling software using the Weibull distribution with the constant α being less than 1.

The reliability function that results from the Weibull distribution is the solution to the differential equation

$$\frac{dR(t)}{dt} = -z(t)R(t) = -\alpha\lambda(\lambda t)^{\alpha-1}R(t)$$

and is given by

$$R(t) = e^{-(\lambda t)^{\alpha}}$$

The expression for $R(t)$ can be verified by calculating the derivative $dR(t)/dt$. Specifically,

$$\frac{dR(t)}{dt} = -e^{-(\lambda t)^{\alpha}}\alpha(\lambda t)^{\alpha-1}(\lambda) = -\alpha\lambda(\lambda t)^{\alpha-1}e^{-(\lambda t)^{\alpha}} = -z(t)R(t)$$

As can be seen, certain values of α result in a reliability function that increases as time increases. For example, if $\alpha = -1$, the reliability is given by

$$R(t) = e^{-1/\lambda t}$$

which approaches 1 as t approaches infinity and is 0 when t is 0. Also note that for $\alpha = 1$, the reliability function is identical to the exponential failure law.

Although time-varying failure rate functions are important in the analysis of software and other systems, by far the most common analysis is performed assuming a constant failure rate function and the exponential failure law. The remainder of this chapter assumes the exponential failure law.

4.2.2 Failure Rate Calculation

An important aspect in the analysis of systems is the estimation of the failure rate of specific components. The most common technique for estimating the failure rate is the United States Department of Defense (USDOD) MIL-HDBK-217 standard ([USDOD 1965], [USDOD 1974], and [USDOD 1979]). Several versions of the standard have been published, including the original standard, MIL-HDBK-217 [USDOD 1965], as well as several revisions, which include, for example, MIL-HDBK-217C [USDOD 1979]. In each version of the standard, the objective has been to develop a model for the failure rate of electronic components using experimental data obtained by analyzing the failures of actual devices. Here, we only summarize the model and the important parameters that are used in calculating the failure rate.

The MIL-HDBK-217B ([Siewiorek and Swarz 1982] and [USDOD 1974]) model predicts the constant failure rate of an integrated circuit (IC) as

$$\lambda = \pi_L \pi_Q (C_1 \pi_T + C_2 \pi_E) \pi_P \quad \text{failures per million hours}$$

where π_L is a *learning* factor, π_Q is a *quality* factor, π_T is a *temperature* factor, π_E is an *environmental* factor, π_P is a *pin* factor, and C_1 and C_2 are *complexity* factors.

The learning factor π_L represents the overall maturity of the fabrication process used to produce the IC. Devices produced using a new, and as yet unproven, manufacturing process are assigned a learning factor of 10, while those produced using a proven process are assigned a learning factor of 1. In other words, the learning factor represents the overall confidence in the ability of the fabrication process to produce devices that will fail infrequently.

The quality factor π_Q represents the amount of device screening that occurs. Device screening is simply the testing that a device goes through prior to being sold by a manufacturer. The lowest level of screening implies that no testing is performed. In other words, the manufacturer simply produces

and sells the IC without verifying that it is operational. At higher levels of screening, the manufacturer randomly selects ICs from a manufacturing run and subjects the selected ICs to certain tests. At even higher levels of screening, the manufacturer thoroughly tests each IC produced. In MIL-HDBK-217B, the quality factor varies from 1 to 300, depending on the level of screening.

The four primary screening levels for ICs are Class A, Class B, Class C, and Class D. Classes A and B are the highest screening levels and are used typically in military applications. The quality factor is 1 for Class A and 2 for Class B. Class C components are representative of high-quality commercial components and have a quality factor of 16. Finally, Class D represents a standard, hermetically sealed commercial component and has a quality factor of 150.

The temperature factor π_T is a function of the device technology, operating temperature, device packaging technology, and power dissipation. The specific equations used for the temperature factor are

$$\pi_T = 0.1e^{(-8121[(1/(T_j+273))-(1/298)])}$$

for linear circuits and

$$\pi_T = 0.1e^{(-4794[(1/(T_j+273))-(1/298)])}$$

for digital bipolar circuits. T_j is the junction temperature and is expressed in degrees Celsius. The second equation given above for π_T is used for transistor-transistor logic (TTL) circuits. As an example, the calculation for a TTL circuit with a junction temperature of 25 degrees Celsius is

$$\pi_T = 0.1e^{(-4794[(1/(25+273))-(1/298)])} = 0.1e^{(0.0)} = 0.1$$

The environmental factor π_E is a function of the harshness of the environment. For example, components operated in an air-conditioned computer room have a much lower environmental factor than those operated on a typical factory floor or in an airborne application. Typical values of the environmental factor vary in the MIL-HDBK-217B standard from 0.2 to 10.0. For example, components located in a computer room have an environmental factor of 0.2; components located in an uninhabited airborne environment have an environmental factor of 6.0; components in a launched missile have an environmental factor of 10.0.

The pin factor π_P is a function of the number of pins on the IC package. In MIL-HDBK-217B, the pin factor ranges from 1.0 to 1.2, for large-scale integration (LSI) technology, as the number of pins increases from 1 to greater than 64. LSI logic is normally defined as an IC having between 100 and 1000 logic gates. For LSI devices, the pin factor is 1.0 if the IC has 25 or fewer pins, 1.1 if the IC has between 26 and 64 pins, and 1.2 for an LSI device having more than 64 pins.

The final factors included in the failure rate model are the complexity factors C_1 and C_2. The complexity factors are a function of the number of gates for logic circuits, the number of transistors for linear circuits, and the number of bits for memories. The complexity factors for ICs having between 100 and 1300 gates are

$$C_1 = (0.0187)e^{(0.00471)(N_g)}$$
$$C_2 = (0.013)e^{(0.00423)(N_g)}$$

where N_g is the number of gates on the IC. The complexity factors for logic having fewer than 100 gates are

$$C_1 = (0.00129)N_g^{(0.677)}$$
$$C_2 = (0.00389)N_g^{(0.359)}$$

where N_g is the number of gates on the IC. The complexity factors for linear circuits are

$$C_1 = (0.00056)N_t^{(0.763)}$$
$$C_2 = (0.0026)N_t^{(0.547)}$$

where N_t is the number of transistors on the IC. The complexity factors for read-only memory (ROM) are

$$C_1 = (0.00114)B^{(0.603)}$$
$$C_2 = (0.00032)B^{(0.646)}$$

where B is the total number of bits in the memory. Finally, the complexity factors for random access memory (RAM) are

$$C_1 = (0.00199)B^{(0.603)}$$
$$C_2 = (0.00056)B^{(0.644)}$$

where B is the total number of bits in the memory.

As an example, consider the calculation of the failure rate for a device having 24 pins and 500 logic gates. We will assume a learning factor of 1.0, a quality factor of 16, a temperature factor of 0.35, and an environmental factor of 0.2. The pin factor is 1.0 for devices with 25 or fewer pins. The complexity factors are calculated as

$$C_1 = 0.0187e^{(0.00471)(500)} = 0.19706$$
$$C_2 = 0.013e^{(0.00423)(500)} = 0.10776$$

and the resulting failure rate is

$$\lambda = \pi_L \pi_Q (C_1 \pi_T + C_2 \pi_E) \pi_P = (1.0)(16)[(0.19706)(0.35) + (0.10776)(0.2)](1.0) = 1.448$$

which is in failures per million hours.

Table 4.1 shows some additional typical values computed using the MIL-HDBK-217B standard.

4.2.3 Mean Time to Failure

In addition to the failure rate, the **mean time to failure (MTTF)** is a useful parameter to specify the quality of a system. The MTTF is the expected time that a system will operate before the *first* failure occurs. For example, if we have N identical systems placed into operation at time $t = 0$, and we measure the time that each system operates before failing, the average time is the MTTF. If each system i operates for a time t_i before encountering the first failure, the MTTF is given by

$$\text{MTTF} = \frac{\sum_{i=1}^{N} t_i}{N}$$

The MTTF can be calculated by finding the expected value of the time of failure. From probability theory, we know that the expected value of a random variable X is

$$E[X] = \int_{-\infty}^{\infty} x f(x)\,dx$$

TABLE 4.1 Typical failure rates calculated using MIL-HDBK-217B ($\pi_L = 1$, $\pi_Q = 16$, $\pi_T = 0.35$, $\pi_E = 0.2$, $\pi_P = 1$)

Number of logic gates	Failure rate (Failures per million hours)
(a) Logic circuits	
50	0.1527
100	0.2312
200	0.3655
500	1.4483
1000	14.4880
(b) Memories (RAM)	
Number of bits	**Failure rate (Failures per million hours)**
1024 (1K)	0.8837
2048 (2K)	1.3491
8192 (8K)	3.1453
16,384 (16K)	4.8033
32,768 (32K)	7.3362

where $f(x)$ is the *probability density function*. In reliability analysis we are interested in the expected value of the time of failure (MTTF), so

$$\text{MTTF} = \int_0^{\infty} t f(t)\,dt$$

where $f(t)$ is the *failure density function*, and the integral runs from 0 to ∞ because the failure density function is undefined for times less than 0. We know, however, that the failure density function is

$$f(t) = \frac{dQ(t)}{dt}$$

so, the MTTF can be written as

$$\text{MTTF} = \int_0^{\infty} t \frac{dQ(t)}{dt}\,dt$$

Using integration by parts and the fact that $dQ(t)/dt = -dR(t)/dt$, we can show that

$$\text{MTTF} = \int_0^{\infty} t \frac{dQ(t)}{dt}\,dt = -\int_0^{\infty} t \frac{dR(t)}{dt}\,dt = [-tR(t) + \int R(t)\,dt]_0^{\infty} = \int_0^{\infty} R(t)\,dt$$

The term $-tR(t)$ clearly disappears when $t = 0$; but, it also disappears when $t = \infty$ because $R(\infty) = 0$. Consequently, the MTTF is defined in terms of the reliability function as

$$\text{MTTF} = \int_0^{\infty} R(t)\,dt$$

which is valid for any reliability function that satisfies $R(\infty) = 0$.

If the reliability function obeys the exponential failure law, the result of calculating the MTTF is given by

$$\text{MTTF} = \int_0^{\infty} e^{-\lambda t}\,dt = \frac{1}{\lambda}$$

In other words, the MTTF of a system that obeys the exponential failure law is the inverse of the failure rate of the system. Note that the reliability at a time equal to the MTTF for the exponential failure law is

$$R(\text{MTTF}) = R\left(\frac{1}{\lambda}\right) = e^{-\lambda(1/\lambda)} = e^{-1} = 0.3678$$

In other words, a system obeying the exponential failure law has a probability of 0.3678 of *not* experiencing a failure before a time equal to the MTTF, given that the system was perfect at the beginning of that time period.

Stated differently, a system obeying the exponential failure law has a probability of 0.6322 of failing during a time period equal to the MTTF, given that the system was perfect at the beginning of that time period.

4.2.4 Mean Time to Repair

The **mean time to repair (MTTR)** is simply the average time required to repair a system. The MTTR is extremely difficult to estimate and is often determined experimentally by injecting a set of faults, one at a time, into a system and measuring the time required to repair the system in each case. The measured repair times are averaged to determine an average time to repair. In other words, if the ith of N faults requires a time t_i to repair, the MTTR is estimated as

$$\text{MTTR} = \frac{\sum_{i=1}^{N} t_i}{N}$$

Often the estimate of the MTTR is improved by averaging over several repair personnel to account for the differences in the abilities of these personnel. For example, if the set of N faults is repaired by M personnel, each of the personnel has an average time to repair, say, MTTR_i, which is the MTTR for the ith person. The estimate of the overall MTTR is the average of the individual MTTRs. In other terms,

$$\text{MTTR} = \frac{\sum_{i=1}^{M} \text{MTTR}_i}{M}$$

The MTTR is normally specified in terms of a **repair rate** μ, which is the average number of repairs that occur per time period. The units of the repair rate are normally number of repairs per hour. The MTTR and the repair rate μ are related by

$$\text{MTTR} = \frac{1}{\mu}$$

4.2.5 Mean Time Between Failure

It is very important to understand the difference between the MTTF and the **mean time between failure (MTBF)**. Unfortunately, these two terms are often used interchangeably. Although the numerical difference is small in many cases, the conceptual difference is very important. The MTTF is the average time until the *first* failure of a system, whereas the MTBF is the average time *between* failures of a system. As noted in the previous section, we can estimate the MTTF for a system by placing each of a population of N

identical systems into operation at time $t = 0$, measuring the time required for each system to encounter its first failure, and averaging these times over the N systems. The MTBF, however, is calculated by averaging the time between failures, including any time required to repair the system and place it back into an operational status. In other words, each of the N systems is operated for some time T and the number of failures encountered by the ith system is recorded as n_i. The average number of failures is computed as

$$n_{avg} = \sum_{i=1}^{N} \frac{n_i}{N}$$

Finally, the MTBF is

$$\text{MTBF} = \frac{T}{n_{avg}}$$

In other words, the MTBF is the total operation time T, divided by the average number of failures experienced during the time T.

If we assume that all repairs to a system make the system perfect once again just as it was when it was new, the relationship between the MTTF and the MTBF is as illustrated in Fig. 4.2. Once successfully placed into operation, a system operates, on the average, a time corresponding to the MTTF before encountering the first failure. The system then requires some time, MTTR, to repair the system and place it back into operation once again. The system then is perfect once again and will operate for a time corresponding to the MTTF before encountering its next failure. The time between the two failures is the sum of the MTTF and the MTTR and is the MTBF. Thus, the difference between the MTTF and the MTBF is the MTTR. Specifically, the MTBF is given by

$$\text{MTBF} = \text{MTTF} + \text{MTTR}$$

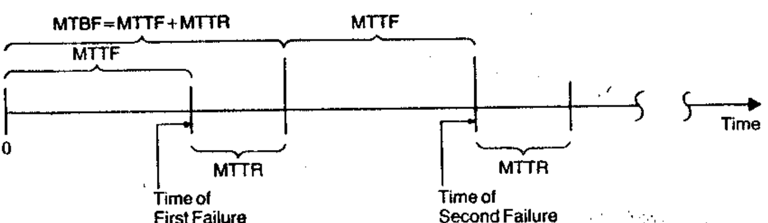


Fig. 4.2 Relationship between the MTBF and the MTTF.

In most practical applications the MTTR is a small fraction of the MTTF, so the *approximation* that the MTBF and MTTF are equal is often quite good. Conceptually, however, it is crucial to understand the difference between the MTBF and the MTTF.

4.2.6. Fault Coverage

An extremely important parameter in the design and analysis of fault-tolerant systems is **fault coverage.** The fault coverage available in a system can have a tremendous impact on the reliability, safety, and other attributes of the system. There are several types of fault coverage, depending on whether the designer is concerned with fault detection, fault location, fault containment, or fault recovery. In addition, there are two primary definitions of fault coverage: one is intuitive, the other is more mathematical.

The intuitive definition is that *coverage* is a measure of a system's ability to perform fault detection, fault location, fault containment, and/or fault recovery. The four primary types of fault coverage are fault detection coverage, fault location coverage, fault containment coverage, and fault recovery coverage. **Fault detection coverage** is a measure of a system's ability to detect faults. For example, a system requirement may be that a certain fraction of all faults be detected; the fault detection coverage is a measure of the system's capability to meet such a requirement. **Fault location coverage** is a measure of a system's ability to locate faults. Once again, it is very common to require a system to locate faults to within easily replaceable modules, and the fault location coverage is a measure of the success with which fault location is performed. **Fault containment coverage** is a measure of a system's ability to contain faults; specifically, the fault containment coverage represents a system's ability to make the *extent* attribute of faults *local* instead of *global*. Finally, **fault recovery coverage** is a measure of a system's ability to recover from faults and maintain an operational status. Clearly, a high fault recovery coverage requires high fault detection, location, and containment coverages.

In the evaluation of fault-tolerant systems, the fault recovery coverage is the most commonly considered, and the general term "fault coverage" is often used to mean fault recovery coverage. In other words, fault coverage is interpreted as a measure of a system's ability to successfully recover after the occurrence of a fault, therefore tolerating the fault. Therefore, when using the term "fault coverage," make sure that the type of coverage—detection, location, containment, or recovery—is understood.

The remainder of this chapter uses the term "fault coverage" to imply fault recovery coverage since fault recovery is the most common form of coverage encountered. In all cases, however, it will be made clear whether detection, location, containment, or recovery coverage is being considered.

Fault coverage is mathematically defined as the conditional probability that, given the existence of a fault, the system recovers [Bouricius, Carter, and Schneider 1969]. In mathematical terms, fault coverage is written as

$$C = P(\text{fault recovery} \mid \text{fault existence})$$

where C is the fault coverage and P(fault recovery | fault existence) is read as the probability of fault recovery *given* the existence of a fault. Recall that fault recovery is the process of maintaining or regaining operational status after a fault occurs.

The fundamental problem with fault coverage is that it is extremely difficult to calculate. Probably the most common approach to estimating fault coverage is to develop a list of all the faults that can occur in a system and to form, from that list, a list of faults that can be detected, a list of faults that can be located, a list of faults that can be contained, and a list of faults from which the system can recover. The fault detection coverage factor, for example, is then computed as simply the fraction of faults that can be detected; that is, the number of faults detected divided by the total number of faults. The remaining fault coverage factors are calculated in a similar manner. As an example, consider the circuit shown in Fig. 4.3 which has fifteen potential sites of stuck-at-1 or stuck-at-0 faults; consequently, there are a total of 30 faults. Table 4.2 shows the input combinations that yield erroneous outputs when certain faults are present, therefore detecting the faults. Note that the circuit performs correctly even if a single stuck-at-0 fault on one of the lines F, G, or M occurs. In other words, a single stuck-at-0 fault on line F, G, or M cannot be detected. As a result, the fault detection coverage for the circuit of Fig. 4.3 is (30–3)/30, or 0.9. In other words, 90% of the stuck-at-1 and stuck-at-0 faults are detected by at least one of the input combinations.

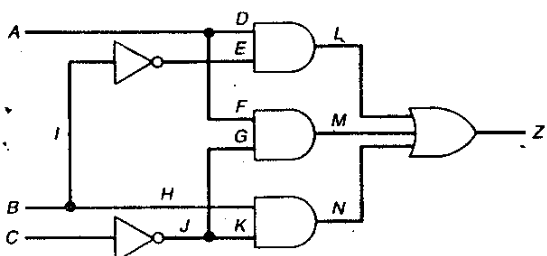


Fig. 4.3 Combinational circuit to illustrate fault detection coverage.

TABLE 4.2 Input patterns capable of detecting faults (test vectors) in the circuit of Fig. 4.3

Fault	Number of test vectors	Test vectors ABC
A_0	2	100, 101
A_1	2	000, 001
B_0	2	010, 111
B_1	2	000, 101
C_0	2	011, 111
C_1	2	010, 110
D_0	1	101
D_1	2	000, 001
E_0	1	101
E_1	1	111
F_0	0	—
F_1	2	000, 101
G_0	0	—
G_1	1	111
H_0	1	010
H_1	1	000
I_0	1	111
I_1	1	101
J_0	2	010, 110
J_1	2	011, 111
K_0	1	010
K_1	2	011, 111
L_0	1	101
L_1	4	000, 001, 011, 111
M_0	0	—
M_1	4	000, 001, 011, 111
N_0	1	010
N_1	4	000, 001, 011, 111
Z_0	4	010, 100, 101, 110
Z_1	4	000, 001, 011, 111

Several important points should be made about the estimation of coverage. First, the estimation of fault coverage requires the definition of the types of faults that can occur. Stating that the fault detection coverage is 0.9, for example, is meaningless unless the types of faults considered are identified. For example, the fault detection coverage for the circuit of Fig. 4.3 is 0.9 for all stuck-at-1 and stuck-at-0 faults, but the fault detection coverage may decrease substantially if stuck-open faults are included.

A second important point about the fault coverage is that it is typically assumed to be a constant. It is easy to envision applications in which the probability of detecting a fault, for example, increases as a function of time, after the occurrence of the fault. However, to simplify the analysis, the various fault coverages are normally assumed to be constants.

4.3 Reliability Modeling

Reliability is perhaps one of the most important attributes of systems. Almost all specifications for systems mandate that certain values for reliability be achieved and in some way proved. We have seen in the previous sections that reliability can be determined experimentally if a set of N systems is operated over a period of time and the number of systems that fail during that time period is recorded. One problem with the experimental approach is the number of systems that would be required to achieve a level of confidence in the experimental results. This is particularly a problem when costs limit the number of systems that can be built. For example, the space shuttle program could not afford to build 1000 of its on-board processing systems such that reliability could be experimentally verified.

A second problem with the experimental approach is the time required to run such experiments. Many systems today are being designed to achieve reliabilities of 0.9_7, or higher, after ten hours of operation. Using the exponential failure law, a reliability of 0.9_7 corresponds to a failure rate of 10^{-8} failures per hour. Therefore, on the average, we would have to wait approximately 100 million hours, or approximately 11,416 years for the first failure to occur. Clearly, we need alternatives to the experimental approach.

The most popular reliability analysis techniques are the analytical approaches. Of the analytical techniques, combinatorial modeling and Markov modeling are the two most commonly used approaches.

4.3.1 Combinatorial Models

Combinatorial models use probabilistic techniques that enumerate the different ways in which a system can remain operational. The probabilities of the events that lead to a system being operational are calculated to form an estimate of the system's reliability.

The reliability of a system is generally derived in terms of the reliabilities of the individual components of the system. The two models of systems that are most common in practice are the series and the parallel. In a **series system**, each element of the system is required to operate correctly for the system to operate correctly. In a **parallel system**, on the other hand, only one of several elements must be operational for the system to perform its functions correctly.

In practice, systems are typically combinations of series and parallel subsystems. Once we have discussed both the series and parallel structures,