

## Chapter 1

# Fundamental Concepts in Fault Tolerance and Reliability Analysis

This chapter introduces the main issues related to the design and analysis of fault-tolerant systems. Our coverage starts with an introduction to faults and their characterization. We then discuss the main issues related to redundancy, including hardware, software, time, and information redundancies. The chapter also covers the fundamental issues related to reliability modeling and evaluation. In particular, the combinatorial reliability model is discussed in details.

### 1.1 Introduction

Computer systems are becoming complex in both their design and architecture. It is not unusual to have a computer system that consists of hundreds and maybe thousands of interacting software and hardware components. Computer systems are developed over a period of time. They usually go through a number of phases (stages) starting from the specification phase, through the design, prototyping, and implementation phases and finally the installation phase. A fault can occur during one or more of these phases. A *fault* is defined as a physical defect that takes place in some part(s) of a system. A fault that occurs during one development stage can become apparent only at some later stage(s). Faults manifest themselves in the form of *error(s)*. When an error is encountered during the operation of a system, it will lead to a *failure*. A system is said to have failed if it cannot deliver its intended function. Figure 1.1 shows a simple example that illustrates the three terms.

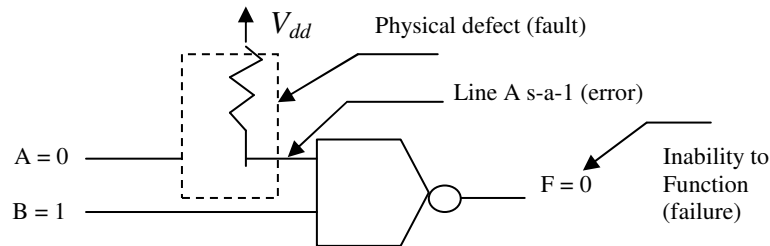
**Example**

Fig. 1.1: The relationship among fault, error, and failure.

Faults can be characterized based on a number of attributes. These include the cause, duration, nature, extent, and value. Table 1.1 summarizes the relationship between these attributes and the type of faults.

Table 1.1: Fault characterization.

Cause	Duration	Nature	Extent	Value
Specification Mistake	Permanent	Hardware	Global	Determinate
Design Mistake	Transient	Software	Local	Indeterminate
Implementation Mistake	Intermittent			
Component Defects				
External Disturbance				

A fault is said to be *permanent* if it continues to exist until it can be repaired. Software bugs are classified as permanent faults. A *transient fault* is one that occurs and disappears at an unknown frequency. A lightning hitting a transmission line causes a transient fault. An *intermittent* fault is one that occurs and disappears at a frequency that can be characterized. A loose contact due to bad soldering can cause an intermittent fault. Fault causes can lead to either software or hardware errors. These in turn can lead to a failure of the system in delivering its intended function.

Based on the level targeted, different techniques can be used to deal with fault(s). These include fault *avoidance*, fault *masking*, and fault *tolerance*. Fault avoidance refers to the techniques used to prevent the occurrence of faults, e.g. quality control (design review, component screening, testing, etc.) and shielding from interference (radiation, humidity, heat, etc.). Fault masking refers to the techniques used to prevent faults from introducing errors, e.g. error correcting codes, majority voting, etc. A fault-tolerant system is a system that continues to function correctly in the presence of hardware failures and/or software errors. A typical fault-tolerant system shall include the following attributes: Fault detection, location, containment, and recovery.

In dealing with faults, a method is needed to model their effect(s). A fault model is a logical abstraction that describes the functional effect of the physical defect. Fault modeling can be made at different levels (ranging from the lowest physical geometric level, up through the switch level, gate level and finally to the functional level). The lower the level of modeling, the more computationally intensive (expensive) the method needed for detecting such fault. The higher the level for fault modeling, the less accurate it is in representing the actual physical defect. The stuck-at gate-level fault model represents a tradeoff between cost and accuracy. It is not as accurate as, but it is less expensive, compared to the switch level fault model. An example of the switch level fault model is the transistor stuck-open (short).

In a given system, failures due to the occurrence of fault(s) take place at a given rate. A failure rate of a system,  $\lambda$ , is defined as the expected number of failures of the system per unit time.

The *Reliability*,  $R(t)$ , of a system is defined as the conditional probability that the system operates correctly throughout the interval  $[t_0, t]$  given that it was operating correctly at  $t_0$ . A relationship between reliability and time that has been widely accepted is the exponential function, i.e.,  $R(t) = e^{-\lambda t}$ . The *mean-time-to-failure (MTTF)* of a system is defined as the expected time that a system will operate before the *first* failure occurs. The relationship between the *MTTF* of a system and its failure rate  $\lambda$  can be expressed as  $MTTF = \frac{1}{\lambda}$ .

Whenever a given system fails, it may be possible to bring it back to function through a process called *system repair*. Repair of a system requires the use of a workshop characterized by a repair rate,  $\mu$ . The repair rate is defined as the average number of repairs that can be performed per unit time. The *mean-time-to-repair (MTTR)* of a system is defined as the expected time that the system will take while in repair (being unavailable). The relationship between the *MTTR* of a system and the repair rate  $\mu$  can be expressed as  $MTTR = \frac{1}{\mu}$ . During its operation, a system can be either available or unavailable (in repair). The *Availability* of a system,  $A(t)$ , is defined as the probability that the system is operating correctly at instant  $t$ . The relationship between the steady state availability of a system,  $A_{ss}$ , and the *MTTF* and the *MTTR* can be expressed as  $A_{ss} = \frac{MTTF}{MTTF + MTTR}$ .

The expected level of service delivered by a given system can be specified in terms of the extent to which the service is offered. When no service degradation can be tolerated, e.g. space missions, a *highly reliable* system is demanded. When short service degradation can be tolerated, e.g. banking systems, a *highly available* system is demanded.

In addition to reliability and availability, a number of other attributes can be used to characterize a given system. Among these *maintainability* and *Safety* are defined next. We define *maintainability*,  $M(t)$ , as the probability that a failed system will be restored to operation within time  $t$ . The relationship between *maintainability* and time can be expressed as  $M(t) = 1 - e^{-\mu t}$ . On the other hand, the *safety* of a system,  $S(t)$ , is defined as probability that the system either performs correctly or discontinues without disturbance to other systems.

## 1.2 Redundancy Techniques

In order for a system to deliver its expected service in the presence of errors caused by faults, some extra (redundant) resources are needed. *Redundancy* can be included in any, or a composite, of the following forms.

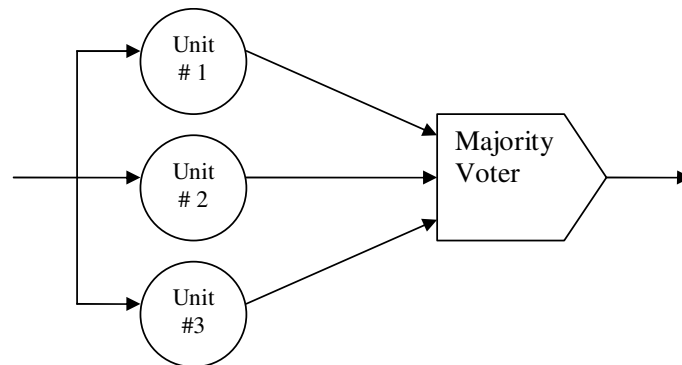


Fig. 1.2: Static HWR (TMR).

### 1.2.1 Hardware Redundancy

This refers to the inclusion of some extra hardware such that concurrent computations can be voted upon, errors can be masked out, or duplicate (spare) hardware can be switched automatically to replace failed components. Three forms of hardware redundancy can be identified. These are Static (Passive), Dynamic (Active) or Hybrid redundancy. In view of Fig. 1.2, Static redundancy requires fault masking; Dynamic redundancy requires fault detection plus replacement of faulty module(s), while Hybrid redundancy requires both fault masking and replacement.

#### 1.2.1.1 Passive (Static) Hardware Redundancy

According to this technique, the effects of faults are essentially masked with no specific indication of their occurrence, i.e., these effects are hidden from the rest of the system. A representative of this technique is demonstrated via the use of the *N-Modular Redundancy* (NMR). Figure 1.2 shows the basic arrangement for a *Triple-Modular Redundancy* (TMR). In this case, three identical modules are used. They perform the same computation at the same time. Their outputs are fed to a majority voter. The output of the voter will be correct if at least two of its inputs

are correct. If at most one module produces incorrect output, the output of the voter will still be correct. The incorrect output is therefore hidden from the rest of the system. If more than one module is expected to be faulty (producing incorrect output), then more modules will have to be included. In general, if  $k$  faulty modules are expected, then  $2k+1$  modules will have to be included. It is also assumed that the voter is perfect. Static hardware redundancy techniques are characterized by being simple, but expensive. They provide uninterrupted service in the presence of faults.

#### 1.2.1.2 Active (Dynamic) Hardware Redundancy

This technique involves removal, or replacement, of the faulty unit(s) in the system, in response to system failure. The process is usually triggered either by internal error detection mechanisms in the faulty unit(s) or by detection of errors in the output(s) of these units. Figure 1.3 shows the basic arrangement for the *standby sparing*. In this arrangement, each unit is provided to the input of an *N to 1 Switch* together with a faulty/not-faulty indication. This fault indication can be obtained by using a duplex system. A duplex system uses two identical modules and a comparator. The comparator checks the output of the two modules and indicates the existence of a fault whenever the two outputs are different.

The final output of the switch can be the output of any of the available units as long as the output of that unit is accompanied by a not-faulty indication, call that the *Primary Unit* (PU). As soon as the accompanied signal to the output of the PU indicates a faulty output, the switch will stop routing the output of that unit to the system output and switch to one of the other units, call them *Spare Units* (SUs). The process is repeated until all SUs are exhausted. *Hot Standby Sparing* refers to the case whereby spare units are powered, operate in Synchrony with the on-line unit(s) and is ready to take over at any times. *Cold Standby Sparing* refers to the case whereby spares are powered only when needed.

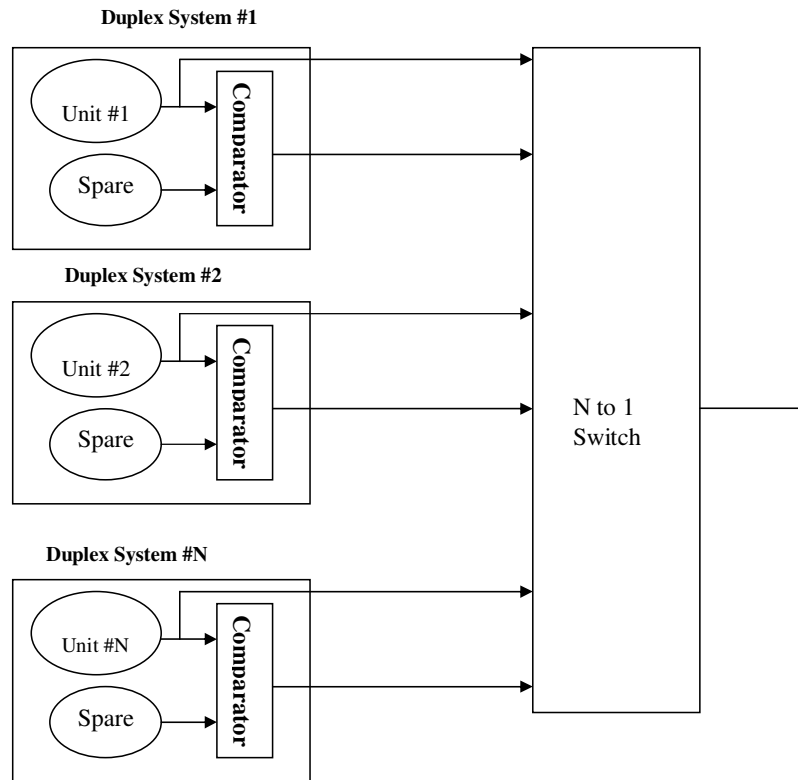


Fig. 1.3: Standby dynamic HWR.

During the switching time, no output should be provided at the output of the system. This system output interruption is a characteristic of the dynamic hardware redundancy techniques. The output of the system is interrupted and no output will be available. Dynamic hardware redundancy techniques are characterized as being inexpensive, but are only suitable for systems that can be interrupted for short time periods.

#### 1.2.1.3 Hybrid Hardware Redundancy

This technique combines the advantages of the passive and the active redundancy. Figure 1.4 shows the basic arrangement for the hybrid hardware redundancy based on the TMR system.

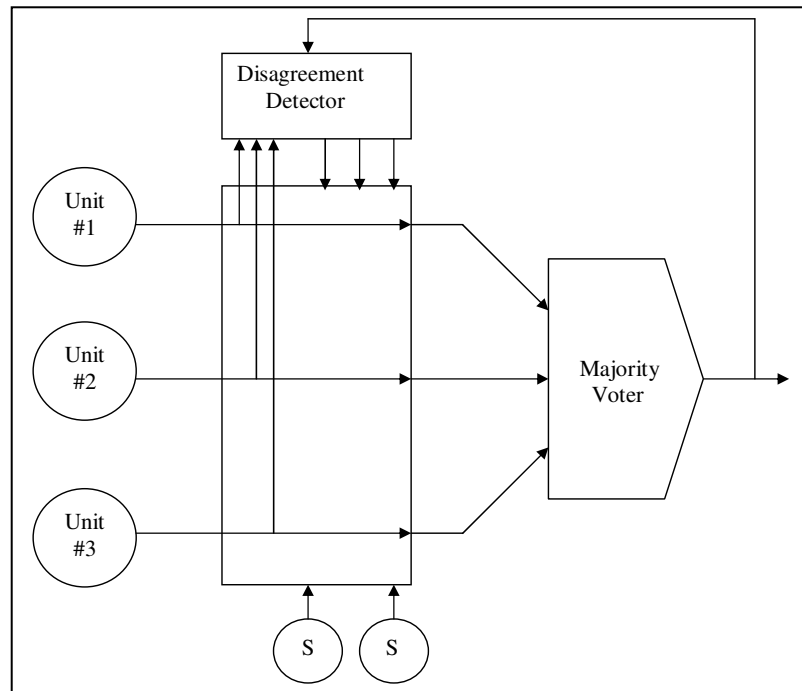


Fig. 1.4: A (3, 2) Hybrid hardware redundancy.

In this arrangement, when one of the primary units (there are three in the figure) fails, it is replaced by a spare (there are two in the figure), so that the basic *TMR* configuration can continue. Identification of the failed unit is made by feeding the output of the majority voter back to a disagreement detector whose job is to compare the voter's output with the outputs produced by the three primary units. Any disagreement with a PU output will indicate that this unit should be replaced by a spare. This replacement is made while the system is not interrupted, i.e., the correct output is still routed to the rest of the system.

In the general case, where  $N$  primary units are used and  $K$  spare units are included, called  $(N, K)$  system, the system can tolerate the failure of up to  $K + \left\lfloor \frac{N}{2} \right\rfloor$  units. The advantage of this system is that it hides the effect of faults (fault masking) while replacing faulty units with spare

ones (reconfiguration). This system is far more expensive than the static hardware redundancy technique. The system should be used in critical applications, such as space applications.

### 1.2.2 Software Redundancy

This refers to the use of extra code, small routines or possibly complete programs, in order to check the correctness or the consistency of the results produced by a given software. A number of software techniques are used. Some of these techniques have their hardware counterpart techniques.

#### 1.2.2.1 Static Software Redundancy Techniques

1. **N-version programming (NVP):** The idea behind this technique is to independently generate  $N > 2$  functionally equivalent programs “VERSIONS” for the same initial specification of a given task. This is usually achieved by having  $N$  individuals (or groups) that work independent to achieve the stated specifications in terms of the algorithms and programming languages used. In the case of 3VP, majority voting is performed on the results procedure by 3 independent programs. The assumption is that the original specification provided to the three programming teams is not flawed. NVP is similar to the hardware majority voting. NVP is expensive, difficult to maintain, and its repair is not trivial.
2. **Transactions:** Here the software is treated as a transaction. Consistency checking is used at the end of the transaction. If the conditions are not met, then restart. The transaction should work the second time around. For a given computation, consistency checking can be performed by inserting assertions to check the results of the computations. As long as the assertion is true, no fault has occurred; otherwise a fault is detected.

3. **Ad-Hoc Techniques:** These are techniques that are application-dependent. For example, a consistency checking on a bank withdrawal can be made by checking that the amount of money withdrawn from any bank machine does not exceed a certain known a priori amount and that the aggregate amount withdrawn by any customer during the 24 hours period from all bank machines does not exceed his/her allowed maximum amount per day. In the case of instruction execution, a check can be made to ensure that no attempt is made by the processor to execute any of the  $(2^n - 2^k)$  invalid instructions (assuming that  $k$  bits are actually used out of the available  $n$  bits op-code). A processor trying to execute any of the invalid instructions is called a *Run Away Processor*. Yet another example for consistency checking is to compare obtained performance with predicted performance based on an accurate model.

#### 1.2.2.2 Dynamic Software Redundancy Techniques

1. **Forward Error Recovery:** In this case, the system will continue operation with the current system state even though it may be faulty. Real-time applications with data collected from a sensor can tolerate missing response to the sensor input.
2. **Backward Error Recovery:** Use previously saved correct state information at the starting point after failure. A copy of the initial data (database, disk, files, etc.) is stored as the process begins. As the process executes, it makes a record of all transactions that affect the data, this is called *Journalizing*. Some subset of the system state is saved at specific points (*check points*) during execution, including data, programs, machine state, etc., that is necessary to the successful continuation and completion of the process past the check point. Rollback is used to recover from failure after repair. This technique is called *Check-pointing*. The primary process records its state on a duplex storage module. When the secondary takes over, it starts by reading the recorded status. This is called checkpoint-restart

technique. The main disadvantage is the need for long repair time, i.e., the time needed to read the status. The use of Checkpoint-restart will yield a highly reliable, but not a highly available system. Alternatively, the primary process can send its status in the form of messages to the secondary process. When the secondary process takes over, it gets its current state from the most recent message. This is called checkpoint-message technique. The main disadvantage of the checkpoint-message is the danger resulting from missing any, or some, of the sent messages to the secondary process. A seemingly more appropriate techniques is called *Persistent*. According to this technique, all state changes are implemented as transactions. When the Secondary takes over, it starts in the null state and has the transaction mechanism undo any recent uncommitted state changes.

3. **Use of Recovery Blocks:** Use of three software elements:
  1. A primary routine to execute critical software functions
  2. An acceptance test routine, which tests the output of the primary routine after each execution.
  3. An alternate routine which performs the same as the primary routine (maybe less efficient or slower) and is invoked by the acceptance test upon detection of failure.
  4. For real-time programs, the Recovery Block should incorporate a watchdog timer to initiate Q if P does not produce acceptable result(s) within the allocated time.

#### 4. **Some Other Software Fault Detection Techniques**

A number of software faults can be detected using techniques that are similar to those used to detect hardware faults. One of the techniques that have been widely used is the *Watchdog Timers*.

According to this technique, a watchdog daemon process is used. The task of the daemon process is to check the status of a given application, i.e., whether the application is alive. This is achieved by periodically sending a signal to the application and checking the return value.

### 1.2.3 Information Redundancy

This refers to the addition of redundant information to data in order to allow fault detection, fault masking, or possibly fault tolerance. Examples of added information include error-detecting and error-correcting codes that are usually added to logic circuits, memories, and data communicated over computer networks. In presenting the basic issues related to information redundancy, we will make use of a number of definitions and theorems. These are presented below.

The term *Code Word* (CW) is used to mean a collection of symbols representing meaningful data. Symbols are grouped according to a predefined set of rules. The term *Error Detecting Code* (EDC) is used to indicate a code that has the ability to expose error(s) in any given data word. Exposure of an erroneous data word is achieved by showing the invalidity of the decoded data word. Based on the principles discussed above, an EDC can be used to initiate a reconfiguration (replacement) process. The term *Error Correcting Code* (ECC) is used to indicate a code that has the ability that if an error has been detected, then it is possible to determine what the correct data would have been. An ECC can be used for masking the effect of errors. Two code words can be distinguished based on their *Hamming Distance* (HD), defined as the number of bits in which the two words differ. For example, the HD between the two code words 0010 and 0101, abbreviated as HD (0010,0101), is  $HD(0010,0101) = 3$ . It should be noted that a  $HD = 1$  between two code words, will mean that one word can be changed into the other by flipping one bit. The term *Code Distance* (CD), of a given set of code words, is used to mean the minimum *HD* between any two code words in that set. Notice that for a given set of code words, if the  $CD = 2$ , then any single bit error will change a code word into an invalid

code word (the minimum distance becomes 1 and not 2). A code having  $CD = 3$  has the ability to expose any single or any double error, i.e., any single or double error will be detected. However, only a single error can be corrected. In general, we can state that in order to detect  $d$  errors, then  $CD \geq (d + 1)$ , to detect and correct  $t$ , or fewer errors, then  $CD \geq 2t + 1$ , and to detect  $d$  and correct  $t$ , or fewer errors, then  $CD \geq t + d + 1$ , where  $t \leq d$ .

Consider, for example, a code consisting of the three code words {10101, 01001, 01110}. This code has a code distance  $CD = 3$ . If, due to errors, in a word-oriented communication system the word (01100) is received, it will be concluded that there are errors because 01100 is not a valid code word. Since  $CD = 3$ , then any single error can be corrected. One way to correct this erroneous word is to EXOR it with each of the valid code words, i.e.,  $01100 \oplus$ 

$$\left\{ \begin{array}{l} 10101 = 11001 \rightarrow HD = 3 \\ 01001 = 00101 \rightarrow HD = 2 \\ 01110 = 00010 \rightarrow HD = 1 \end{array} \right.$$

We see that  $HD(01100, 01110) = 1$ , so the originally transmitted word must have been 01110.

Table 1.2 shows the ED and EC ability for three different example codes.

Table 1.2: The ED and EC ability for three different example codes.

Code	Code distance	Detection ability	Correction ability
{00, 11}	2	1	0
{000, 111}	3	2 or fewer	1
{0000, 1111}	4	{3} or {2 or fewer}	{0} or {1}

It should be noted that the code {000, 111} can detected 2 or fewer errors and correct none or can correct only one error, but not both. Similarly, the code {0000, 1111} can detect 3 errors and correct none or alternatively it can detect 2 or fewer errors and correct only one error.

In addition to the detection/correction ability of a code, i.e., the CD of a code, codes can be characterized based on a property called

*separability*. If the data information and the detection/correction information are separable, the resulting code is called a *separable code*; otherwise the code is *non-separable*. We will introduce a number of codes, starting with error detecting codes and followed by error correcting codes.

### 1.2.3.1 Error Detecting Codes

These codes usually have less overhead than the error correcting codes but they lack any error correcting capability. We will discuss four different techniques for error detection: Parity codes, Borden codes, Berger code and Bose Code.

#### (1) Parity Codes

*Single bit odd (even) parity*: Addition of one extra bit such that the total number of 1's is odd (even). This simple code has applications in data storing/retrieving in/from computer main memory. The basic scheme used for this is shown in Fig. 1.5.

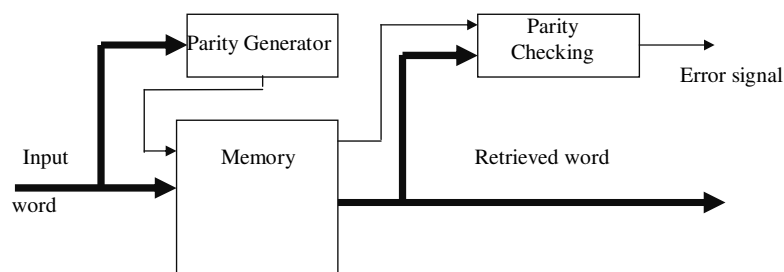


Fig. 1.5: Parity checking in computer main memory.

In this case, before storing a word in the memory, a parity generator is used to compute the parity bit required. Then both the original word and the computed parity bit are stored in the memory. On retrieval, a parity checker is used to compute the parity based on the data bits stored. The computed parity bit and the stored parity bit are compared by the

parity checker and an error signal is set accordingly, i.e., if the computed and stored parity bits match, then the retrieved word is assumed to be correct; otherwise the retrieved word is assumed to be in error. It should be noted that for  $n$  bit data, there are  $2^n$  possible data words. The addition of 1 parity bit will result in a total of  $2^{n+1}$  possible words. Among these there is  $\frac{2^{n+1}}{2}$  words with odd number of 1s and  $\frac{2^{n+1}}{2}$  words with even number of 1s. For odd (even) parity, only those words with odd (even) number of 1's are valid code words. In the presence of a single error, an odd (even) parity code word will change into an even (odd) parity and thus, the error will be detected. Although simple, but the simple parity code has limited fault detection ability, i.e., it can only detect the occurrence of single errors. The occurrence of double errors will escape the detection unobserved. In general, simple parity codes have the ability to detect the occurrence of only odd number of errors. The occurrence of even number of errors will not be detected by simple parity codes.

## (2) Borden Codes

Borden code is a general formulation for a wide range of error detecting codes. In general, Borden codes are non-separable error detecting codes. They are characterized by the following definition:

### Definition

If  $C_{mn}$  is the set of codes of length  $n$  for which exactly  $m$  bits are ones, then the union of all such codes with  $m$  being the set of values congruent to  $\lfloor n/2 \rfloor \bmod (d + 1)$  is known as the Borden ( $n, d$ ) code.

**Example:** Suppose that we would like to construct the Borden (5,2) code. We first have to compute the quantity  $\lfloor n/2 \rfloor \bmod (d + 1) = \lfloor 5/2 \rfloor \bmod (3) = 2$ . This means that  $m$  belongs to the set  $\{0,2,4\}$ . Thus any word of length 5, of which either no bits, two bits, or four bits are 1 belongs to this Borden code. Here are all the possibilities:  $\{00000, 00011, 00110, 01100, 11000, 00101, 01010, 10100, 01001, 10010, 10001, 11110, 11101, 11011, 10111, 01111\}$

The Borden  $(n, d)$  can detect  $d$  unidirectional errors (errors that cause either a  $0 \rightarrow 1$  or  $1 \rightarrow 0$  transition, but not both). It is the optimal code for all  $d$ -unidirectional error detecting codes. It is also interesting to observe the following special cases:

If  $d = 1$ , the Borden code becomes the well-known parity code.

If  $n = 2d$ , we get a special kind of code called the  $k/2k$  (or  $k$  out of  $2k$ ) code.

### (3) Berger Code

The Berger error detecting code is a separable code capable of detecting all unidirectional errors. It is formulated by appending check bits to the data word. The check bits constitute the binary representation of the number of 0's in the data word. For example, for a 3 bit long data word, we need 2 bits for the check. Table 1.3 shows the valid Berger code words.

Table 1.3: Valid Berger code words.

Data word	Check bits	Codeword
000	11	00011
001	10	00110
010	10	01010
011	01	01101
100	10	10010
101	01	10101
110	01	11001
111	00	11100

As can be seen, the Berger code is simpler to deal with than the Borden codes. However, the Bose code is more efficient than the Berger code. This is shown below.

## (4) Bose Code

The Bose code provides the same error detecting capability that the Berger code does, but with fewer check bits. It requires exactly  $r$  extra bits for a  $2^r$  data word. Bose codes are generated according to the following rules:

If the number of zeros in the data word is 0 or  $2^r$ , then complement the first  $2^{r-1}$  data bits and append  $2^{r-1} - 1$  in the check bit locations.

If the number of zeros in the data word is between  $2^{r-1}$  and  $2^r - 1$ , then append the binary representation of this number (as in Berger codes).

If the number of zeros is between 1 and  $2^{r-1} - 1$ , append this number minus one as a check symbol.

Table 1.4 shows an example for constructing Bose code with 4 data bits.

Table 1.4: An example Bose code with 4 data bits.

Data word	Check bits	Codeword
0000	01	110001
0001	11	000111
0010	11	001011
0011	10	001110
0100	11	010011
0101	10	010110
0110	10	011010
0111	00	011100
1000	11	100011
1001	10	100110
1010	10	101010
1011	00	101100
1100	10	110010
1101	00	110100
1110	00	111000
1111	01	001101

### 1.2.3.2 Error Correcting Codes

In this section, we discuss a number of error correcting codes. These are codes that have the ability not only to detect the existence of a fault, but they have also the ability to locate the errors and hence allow for correcting these errors.

#### Hamming Codes

Hamming codes are the earliest linear error correcting codes. They consist of “k” data bits and “r” check bits such that  $k = 2^r - r - 1$ ,  $n = 2^r - 1$ ,  $r \geq 3$ . Hamming codes are usually referred to as Hamming (n, k) codes. Thus, for  $r = 3$ , we have the Hamming (7,4) code. Hamming codes can be used to correct single errors or detect double errors but not both simultaneously.

The parity checker matrix,  $H$ , for the Hamming code can be formed by filling the columns of the  $P$  matrix with all the nonzero binary combinations that do not appear in the columns of the neighboring identity matrix. For example, for Hamming (7,4), the following binary combinations are used for the identity matrix (001,010,100), which leaves us with the following combinations that can be put in any order in the  $P$  matrix (011,101,110,111). Thus, the parity checker matrix may appear as shown below.

$$H = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix}$$

Given the forms of the *generator matrix*  $G$  and the  $H$  matrix, we can construct the  $G$  matrix by mapping the identity matrix and the  $P$  matrix to their correct positions (in the  $H$  matrix, the  $P$  matrix is transposed):

$$G = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Now, to encode a message, we multiply the message by the generator matrix. To decode the received message, we multiply the received vector by the transpose of the parity check matrix. If the syndrome is zero, then the received vector is assumed correct, otherwise the syndrome will be used to access a table containing the error vectors.

**Example:** The following steps show the encoding and decoding of a message according to the Hamming code.

$$\vec{m} = (0011)$$

$$\vec{c} = \vec{m}G = (0 \ 0 \ 1 \ 1) \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} = (1 \ 0 \ 0 \ 0 \ 0 \ 1 \ 1)$$

To decode:

$$\vec{c} = (1000011)$$

$$\vec{s} = \vec{c}H^T = (1 \ 0 \ 0 \ 0 \ 0 \ 1 \ 1) \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} = (0 \ 0 \ 0)$$

Since the syndrome is zero, the codeword is accepted as valid. If, however, an error occurred, say, the received message was (1010011), then we would get a nonzero syndrome as shown below.

$$\vec{c} = (1010011)$$

$$\vec{s} = \vec{c}H^T = (1 \ 0 \ 1 \ 0 \ 0 \ 1 \ 1) \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} = (0 \ 0 \ 1)$$

This syndrome is used to index a table that will output an error vector. This vector is added to the received message to get the corrected message. To construct the syndrome table, we note that an error in a certain column in the received message correspond to adding the corresponding row in  $H^T$ . The Syndrome Table for the Hamming code is shown in Table 1.5.

Table 1.5: Syndrome Table for Hamming code.

Error vector	Syndrome
1000000	100
0100000	010
0010000	001
0001000	110
0000100	101
0000010	011
0000001	111

### 1.2.3.3 SEC-DED Codes

The Hamming codes discussed above can correct a single error or detect double errors, but not both simultaneously. Although it is quite useful in cases where only a single error is of significant probability, it does carry the hazard of miscorrecting double errors. To solve this, we may

introduce an extra parity bit. This parity bit will change the appearance of the  $H$  matrix and the  $G$  matrix. This extra parity bit will be appended to the beginning of the codeword. For a Hamming (7,4) code, the  $H$  matrix would become as shown below.

$$H = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

As can be seen, we added the first column and the last row to the original  $H$  matrix. When we perform the decoding, we transpose  $H$  and the first column becomes the first row. Thus, the first three columns of the syndrome will not be affected by the parity bit. The final column consists of all 1's. This means that now an entire parity check bit will be appended to the syndrome. The parity bit depends on all data bits and parity bits. Thus, it cannot be part of the  $G$  matrix but rather has to be appended manually after performing a regular Hamming encoding.

To perform the decoding of SEC-DED Hamming codes, we separate the resulting syndrome into the regular Hamming syndrome ( $s$ ), and the parity check bit ( $p$ ). Decoding then takes place as shown below.

If  $p = 0$  and  $\bar{s} = 0$ , then no errors have occurred.

If  $p = 1$ , a single error occurred and we can correct it as shown previously.

If  $p = 0$  and  $\bar{s} \neq 0$ , then a double error have occurred and is not correctable.

Using the SEC-DED codes, we are guaranteed that a double error will not be miscorrected but will rather be just reported as a double error. This is very useful in cases where double errors are not too probable but nevertheless have some significant probability.

### Cyclic Codes

The second class of codes we will discuss is the cyclic codes. These are linear codes with an added property. The added property is that a rotation of any valid codeword is also a valid codeword. For example, (00000000, 110110110, 011011011, 101101101) is a valid cyclic code, since it is linear and the rotation criteria holds. This property facilitates the design of decoder circuits.

Cyclic codes can be constructed using the same matrix method that was used for Hamming codes. A closer look at the  $G$  matrix format will reveal that each row of the  $G$  matrix represents a valid codeword, and that all code words are made of combinations of these rows. Thus, if the rows of the generator matrix are rotations of a certain vector, then any combination of these rows can be expressed as rotations of a different combination of rows. The general structure of the  $G$  matrix in this case will become as shown below.

$$G = \begin{pmatrix} g_0 & g_1 & \cdot & \cdot & \cdot & g_{n-k} & 0 & 0 & 0 \\ 0 & g_0 & g_1 & \cdot & \cdot & \cdot & g_{n-k} & 0 & 0 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & 0 & g_0 & g_1 & \cdot & \cdot & g_{n-k} \end{pmatrix}$$

Thus, the characteristics of the  $G$  matrix are all encapsulated in the vector  $g_0g_1\dots g_{n-k}$ . If we represent this vector and the message vector as polynomials, then a very efficient means of representing cyclic codes can be used.

### Polynomial Representation of Binary Vectors

We can represent any binary vector by a polynomial of a degree less than the number of bits of the vector by 1. In this polynomial  $f(x)$ , each term gets as its weight a digit from the binary vector, with the highest power

term getting its weight from the most significant bit of the binary vector, e.g.,  $(11001) = x^4 + x^3 + 1$  and  $(00111) = x^2 + x + 1$ . As in the GF(2) arithmetic, addition can be performed as  $x^a \pm x^a = 0$ . The following two examples illustrate the multiplication and division of polynomials.

$$\begin{aligned}
 A &= (11001) = x^4 + x^3 + 1 \text{ and } B = (00111) = x^2 + x + 1 \\
 A * B &= (x^4 + x^3 + 1)(x^2 + x + 1) = x^6 + x^5 + x^2 + x^5 + x^4 + x + x^4 + x^3 + 1 \\
 &= x^6 + x^3 + x^2 + x + 1 \\
 A/B &= x^2 + x + 1 \overline{) x^4 + x^3 + 1} \\
 &\quad \underline{x^4 + x^3 + x^2} \\
 &\quad \quad x^2 + 1 \\
 &\quad \quad \underline{x^2 + x + 1} \\
 &\quad \quad \quad x \\
 &\Rightarrow A = (x^2 + 1)B + x
 \end{aligned}$$

The idea of cyclic coding involves adding redundancy to the original data vector by multiplying it with the polynomial whose coefficients are  $g_0g_1\dots g_{n-k}$  (discussed earlier). This polynomial is known as the *generator polynomial* and has degree  $r = k-n$ . The product constitutes the codeword and has exactly  $n$  bits (with the polynomial representing it having power less than or equal  $n-1$ ).

If we suppose that the generator polynomial  $g(x)$  is a factor of  $x^n + 1$ . In this case  $x^n + 1 = g(x)h(x)$ . If this criterion is met, then we can mathematically express the encoding and decoding process of the cyclic code as follows:

To encode a message  $m(x)$  into a codeword  $c(x)$ , we multiply the two polynomials:

$$c(x) = m(x)g(x)$$

To decode a codeword  $c(x)$ , we divide the two polynomials:

$$m'(x) = c(x)/g(x)$$

Since the decoded message may differ from the original message, we need some means of error checking. Consider the following modulo multiplication result for a valid codeword:

$$\begin{aligned} c(x)h(x) \bmod(x^n + 1) &= m(x)g(x)h(x) \bmod(x^n + 1) \\ &= m(x)(x^n + 1) \bmod(x^n + 1) = 0 \end{aligned}$$

However, if the codeword is invalid, then one can express the codeword as  $c'(x) = c(x) + e(x)$ , where  $e(x)$  is an error vector. In this case, the modulo multiplication becomes:

$$\begin{aligned} c'(x)h(x) \bmod(x^n + 1) &= c(x)h(x) \bmod(x^n + 1) + e(x)h(x) \bmod(x^n + 1) \\ &= e(x)h(x) \bmod(x^n + 1) \end{aligned}$$

The significance of this result is that the resulting quantity depends only on the error vector and can be thus taken as a syndrome for error correction. The syndrome can then be used to address a lookup table which will reveal the error vector. This method is quite general and can be used in any cyclic code. In the following section, we will look at a more specific case of cyclic code implementation, namely, the *systematic cyclic codes*.

### Generating and Decoding Systematic Cyclic Codes

It is useful to generate a separable version of the cyclic codes. This can be accomplished if we insert parity bits according to some rules such that the resulting codeword maintains its cyclic characteristic. If we agree to append the parity bits at the least significant positions instead of the most significant, then we can get a systematic cyclic code by setting:

$$c(x) = x^r m(x) + x^r m(x) \bmod g(x)$$

The first term is nothing more than a shift by  $r$  bits to accommodate the parity bits. The actual parity bits are the ones represented by  $x^r m(x) \bmod g(x)$ . Note that the encoding here is different from the general

case where we multiply by the generating polynomial. In this case, we construct the parity bits by dividing the shifted message by the generating polynomial and appending the remainder to the message. This division can be realized in hardware by the divider circuit shown in Fig. 1.6.

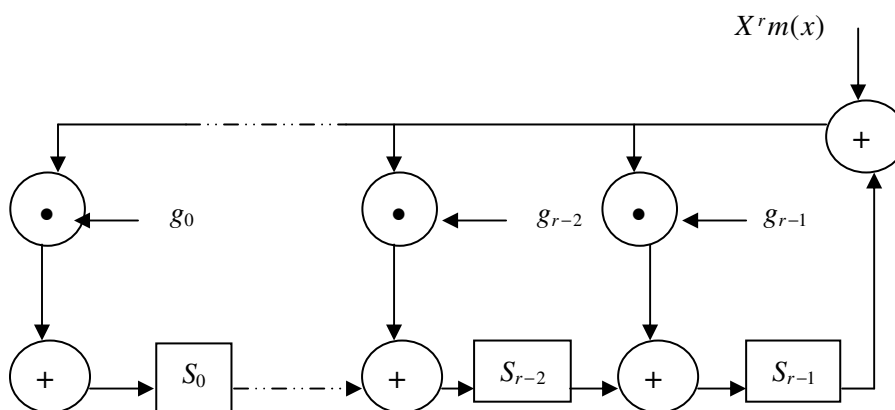


Fig. 1.6: Divider circuit.

In this circuit, the shifted message is passed through the circuit. If the shifted message enters MSB first, then after k steps, the remainder can be taken in parallel from the intermediate latches. The encoded message can be constructed by appending these parity bits to the original message.

The syndrome for this encoding is calculated by dividing a shifted version of the codeword by the generating polynomial. For a valid codeword, this translates mathematically to:

$$x^r c(x) \bmod g(x) = [x^r \bmod g(x)] [x^r m(x) \bmod g(x) + x^r m(x) \bmod g(x)] = 0$$

For an invalid codeword, the resulting syndrome will be:

$$s(x) = x^r c'(x) \bmod g(x) = x^r (c(x) + e(x)) \bmod g(x) = x^r e(x) \bmod g(x)$$

which depends only on the error vector, as in the general non-separable case. Each error vector will have a syndrome associated with it. Thus, a hardware design for the decoder would be possible using a division circuit and a syndrome lookup table. A block diagram of such a design is shown in Fig. 1.7.

$X'V(x)$

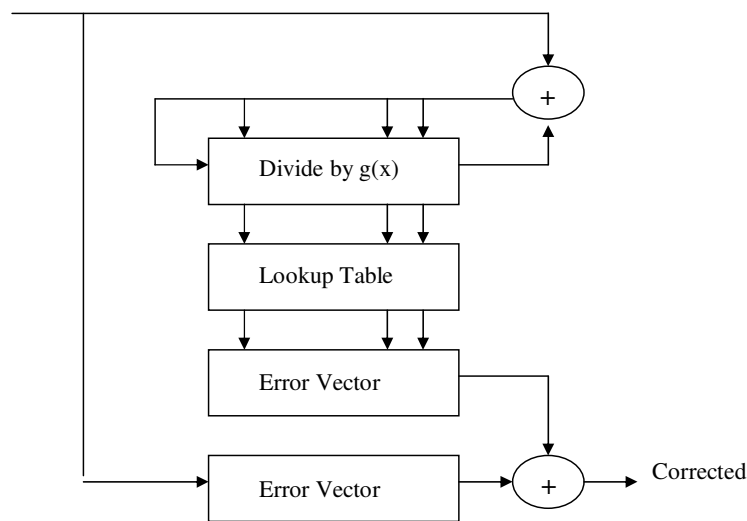


Fig. 1.7: Decoding circuit for systematic cyclic code.

#### 1.2.3.4 CRC Codes

The CRC (Cyclic Redundancy Check) codes are cyclic codes that lack error correction capability but can be used to detect errors. They are often used as complementary circuits for other error correcting circuits. They are mainly used to detect miscorrections of the decoder, as was pointed out previously.

CRC codes are used extensively in microcomputer systems. They are used to detect errors in floppy disks and compact disks. Selecting a certain CRC code requires selecting a generator polynomial. CRC polynomials are usually selected from a family of polynomials called

primitive polynomials. A primitive polynomial is any polynomial  $g(x)$  of degree  $r-1$  that satisfies the following two conditions:

$$g(x) \bmod (x^{2^{r-1}} - 1) = 0$$

$$g(x) \bmod (x^m - 1) \neq 0, \quad m < 2^{r-1} - 1$$

There are various standard CRC polynomials that have been tested to achieve a high error detection capability. These standards are issued by organizations such as IEEE and are adopted by manufacturers for different types of communication media and devices. Table 1.6 shows some of the standard CRC polynomials.

Table 1.6: Some standard CRC polynomials.

Name	Max block length	Degree	Polynomial (in octal)
CRC-24	1023	24	140050401
CRC-32A	1023	32	50020114342
CRC-12	2047	12	14017
CRC-SDLC	16383	16	320227

From: Applied Coding and Information Theory for Engineers, Richard Wells.

### 1.2.3.5 Convolution Codes

Convolution codes differ greatly from the previous encoding methods, perhaps not in the mechanism and implementation but definitely in the use. In all previous cases, the encoder would take blocks of data (of length  $k$ ) and add some redundancy to it. Convolution codes, however, can work on streams of data of different lengths. This is useful when we would like a high code rate ( $k/n$ ) and would also like good error correcting capability. For this reason, convolution codes are generally used in noisy channels with high error rates. Convolution codes are implemented using shift registers that are arranged such as to form a finite impulse response (FIR) circuit. FIR circuits (or channels) have a

finite response to a single input. This is unlike IIR (infinite impulse response) channels, that have an infinite memory of a single input.

FIR circuits can be constructed using shift registers. Convolution codes use these FIR circuits to add a certain amount of redundancy to the input. An example of a convolution encoder is shown in Fig. 1.8.

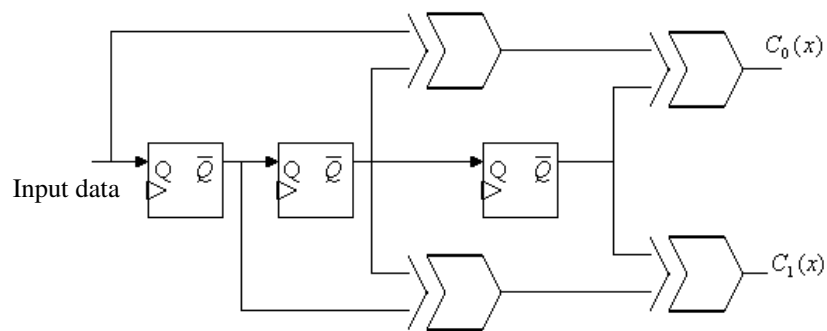


Fig. 1.8: An example of a convolution encoder.

In this figure, the input data stream passes through the register stages shown. The two outputs are interleaved in a certain way and sent. In this case, for each input bit there are two output bits, one coming from  $c_0(x)$  and the other one from  $c_1(x)$ . The code ratio for this encoder ( $k/n$ ), however, is slightly below the expected 0.5 ratio. This is because we need to flush out the contents of the registers when we are done encoding a certain length of data stream (called a frame). The extra bits from the registers are also sent with the code, lowering the code ratio. For this reason, it is good to choose high values for the frame length such that the overhead of sending the flushed values is minimal.

Following is an example of encoding a data stream using the convolution encoder of Fig. 1.8.

**Example 7:** We want to send the following data stream (1001101) using the encoder of Fig. 1.8. We will denote the values on the flip flops by  $S_0$ ,  $S_1$  and  $S_2$ . The following equations and table can be used to determine the outputs  $c_0(x)$  and  $c_1(x)$ :

$$S_0^+ = \text{input}$$

$$S_1^+ = S_0$$

$$S_2^+ = S_1$$

$$c_0 = S_1 + S_2 + \text{input}$$

$$c_1 = S_0 + S_1 + S_2$$

Time	Input	$S_0$	$S_1$	$S_2$	$C_0$	$C_1$
0	?	0	0	0	0	0
1	1	1	0	0	1	0
2	0	0	1	0	0	1
3	1	1	0	1	0	1
4	1	1	1	0	0	0
5	0	0	1	1	1	0
6	0	0	0	1	0	0
7	1	1	0	0	0	1
8	0	0	1	0	0	1
9	0	0	0	1	1	1
10	0	0	0	0	1	1

The three final inputs are not part of the data but are deliberately inserted to flush the contents of the shift register. If the output is interleaved such that the bit from  $c_0$  is sent first, then the output stream would look like this: (10010100100001011111). The code ratio is:  $7/20 = 0.35 < 0.5$

### 1.2.4 Time Redundancy

Time redundancy refers to the repetition of a given computation a number of times and a comparison of the results to determine if a discrepancy exists. The existence of a discrepancy between subsequent computations indicates the existence of transient or intermittent faults. The basic scheme used is shown in Fig. 1.9.

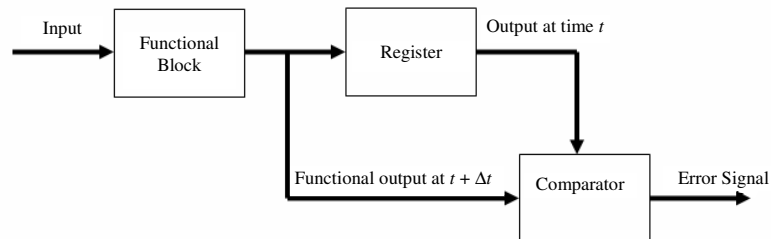


Fig. 1.9: Time redundancy basic scheme.

In this scheme, all intermittent and transient fault occurring in either of the computation steps at times  $t$  and  $t + \Delta t$ , but not both, can be detected. No permanent faults can be detected.

#### 1.2.4.1 Permanent Error Detection with Time Redundancy

A similar arrangement can be used to detect permanent faults in a functional block. Consider, for example, the arrangement shown in Fig. 1.10. According to this arrangement, the computation using the input data is first performed at time  $t$ . The results of this computation is then stored in a register. The same data is used to repeat the computation, using the same functional block at time  $t + \Delta t$ . However, this time the input data is first encoded in some way. The results of the computation is then decoded and the results are compared to the results produced before. Any discrepancy will indicate a permanent fault in the functional block.

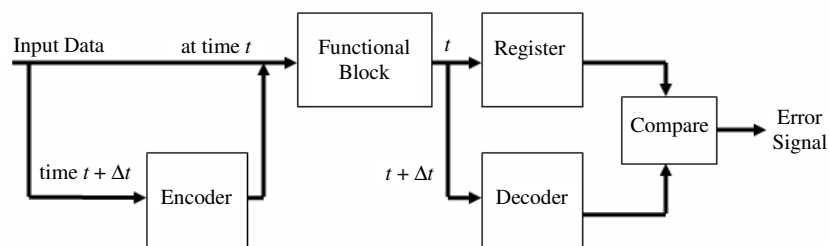


Fig. 1.10: Permanent fault detection scheme.

Assume that the input data is called  $x$ , the functional computation is called  $F$ , the encoding is called  $E$ , and the decoding is called  $D$ , then given that there is no permanent fault in the functional block, we can write the following relation  $D(F(E(x))) = F(x) \forall x$ . If  $D$  and  $E$  are properly chosen such that a failure in  $F$  will effect  $F(x)$  and  $F(E(x))$  differently, thus making the output of time  $t$  and at time  $t + \Delta t$  not equal, then an error signal will be produced. The above condition requires the following to be true:  $D(E(x)) = x$ , i.e.  $D$  and  $E$  are inverse of each other. Consider, for example, the case whereby the encoding of the input data is just the complementation of the input and that the decoding is just the complementation of the output from the functional block, i.e.,  $F(\bar{x}) = \overline{F(x)}$ . This arrangement is shown in Fig. 1.11.

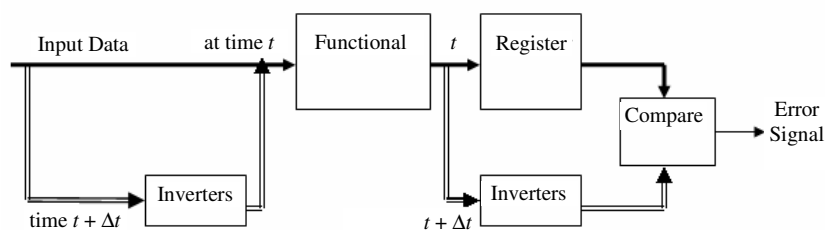


Fig. 1.11: Example encoding/decoding.

The above mentioned property is that of what is known as *self-dual function*. Using this property, it can be seen that for a self-dual function, the application of the input  $x$  will produce an output that is the complement of the output produced if this input is followed by the complement, i.e.,  $\bar{x}$ . Therefore, if the output of the functional block in response to an input  $x$  is  $1(0)$ , then the output of the same functional block in response to an input  $\bar{x}$  will be  $0(1)$ . This is called *alternating logic*. In order to be able to detect the existence of a permanent fault in a functional block that is self-dual using alternating logic is to try to find at least one input combination for which the fault will not produce alternating output.

The above discussion shows that it is possible to detect the existence of a permanent fault using alternating logic only if the function realized by the functional block under test possesses the self-duality property. It is, however, possible to generalize this observation to include those non-self-dual function. This is because of the following general observation: any non-self dual function of  $n$  variables can be converted into a function of  $n + 1$  variable that is self dual and therefore can be realized by an alternating logic circuit.

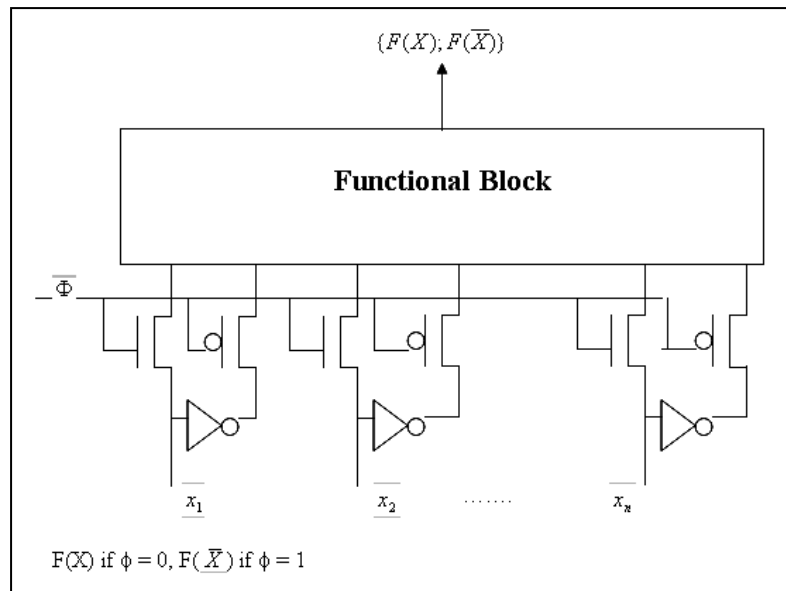


Fig. 1.12: A possible scheme for alternating logic.

The scheme shown in Fig. 1.12 indicates that if the control input  $\Phi = 0$  then the functional block will compute  $F(X)$ , while if  $\Phi = 1$  then the functional block will compute  $F(\bar{X})$  instead.

### Recomputing with Shifted Operands (RESO)

This method uses the basic time redundancy techniques discussed above in achieving concurrent fault detection in arithmetic logic units (ALUs). In this case, shift operations are used as the encoding functions. For example, a shift left operation can be used as the encoding function while a shift right operation can be used as the decoding function. Figure 1.13 shows an illustration of an ALU designed to allow such encoding/decoding to be realized.

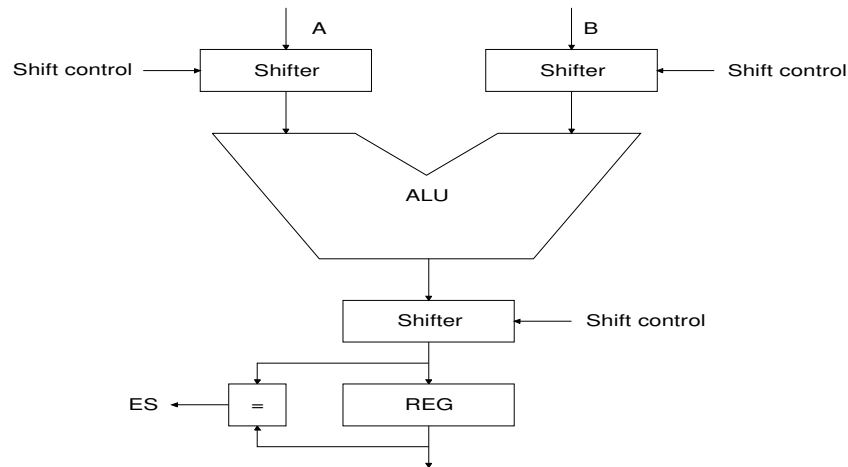


Fig. 1.13: RESO in an ALU.

It should be noted that RESO technique assumes bit-slice organization of the ALU.

### 1.3 Reliability Modeling and Evaluation

Reliability modeling aims at using abstract representation of systems as means for assessing their reliability. Two basic techniques have been used. These are the empirical and the analytical techniques (see Fig. 1.14).

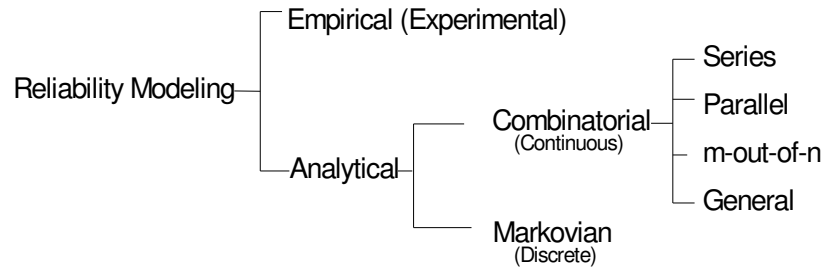


Fig. 1.14: Reliability modeling.

### 1.3.1 Empirical Models

According to this model, a set of  $N$  systems are operated over a long period of time and the number of failed systems during that time are recorded. The percentage of the failed system to the total number of operated systems,  $N$ , is used as an indication of the reliability of the systems. For example, if the systems are operated for 1000 times and out of those 1000 times, there were 100 times in which the systems have failed, then it will be assumed with reasonable accuracy that the reliability of the system is 90%. A number of concerns exist about the accuracy of the empirical model. One such concerns is that the number of systems required to achieve a level of confidence is large. For example, it would have been extremely expensive if a 1000 on-board processing systems of the space shuttle has to be built such that the reliability could be experimentally determined. In addition, the time required for the experiment may be too long to afford. For example, using the exponential failure rate for a system with reliability of 0.97 will mean a failure rate of approximately,  $\lambda = 10^{-8}$  failures per unit time.

### 1.3.2 The Analytical Technique

The analytical model is based on the use of the probability of failure of individual components of a given system in order to arrive at a measure for the probability of failure of the overall system. The overall failure probability of a system will depend not only on the individual failure

probability of its individual components but also on the way these components are interconnected in order to form the overall system. Two techniques exist for computing a system reliability according to the analytical technique. These are the *Combinatorial* (Continuous) and the Markov (Discrete) models. These are discussed below.

(1) Combinatorial (Continuous) Model

According to this technique, the number of ways in which a system can continue to operate, given the probability of failure of its individual components, is enumerated. A number of models exist for the interconnection among the system's components. These are summarized in Table 1.7. Shown also in the table are the corresponding expression of system reliability  $R(t)$  in terms of the reliability of the individual components of the system.

Table 1.7: Reliability models.

Technique	Reliability equation
1. Series	$R_{\text{series}}(t) = \prod_{i=1}^N R_i$
2. Parallel	$R_{\text{parallel}}(t) = 1 - \prod_{i=1}^N (1 - R_i)$
3. Series/Parallel	$R_{\text{sp}}(t) = \prod_{i=1}^N R_{\text{parallel}_i}$ $R_{\text{parallel}_i}$ is the reliability of the $i^{\text{th}}$ group
4. Parallel/Series	$R_{\text{ps}}(t) = 1 - \prod_{i=1}^N (1 - R_{\text{series}_i})$ $R_{\text{series}_i}$ is the reliability of the $i^{\text{th}}$ group
5. M out of N	$R_{\text{MoN}} = \sum_{i=0}^{N-M} \binom{N}{i} (1 - R)^i * R^{N-i}$
6. Exact	$P_{\text{system}} = (P_{\text{system}} A)P_A + (P_{\text{system}} \bar{A})(1 - P_A)$
7. Approximate	$R_{\text{system}} \leq 1 - \prod_{i=1}^n (1 - R_{\text{path}_i})$ $n$ is the number of paths from input to output

In modeling systems according to the combinatorial model, a reliability block diagram (RBD) is formed. The RBD shows the connection among the system's components from the reliability point of view (as opposed to the functionality view point).

We provide some more details on the model presented under exact above. This model is extremely useful in modeling complex systems by de-synthesizing them into simpler ones. The theorem according to which such model is formed is called *Bay's Theorem*. The theorem states that for a given system consisting of 2 or more modules, select one module (any module) and call it module A. Two mutually exclusive events occur. These are

1. The system with A working ( $P_{\text{system}|A}$ )
2. The system with A not working ( $P_{\text{system}|\bar{A}}$ )

The probability of the system working  $P_{\text{system}}$  can then be expressed as shown in the table above, i.e.,  $P_{\text{system}} = (P_{\text{system}|A})P_A + (P_{\text{system}|\bar{A}})(1-P_A)$ , where  $P_A$  is the probability that module A is working while  $(1-P_A)$  is the probability that module A is not working. The following example illustrates the use of the theorem.

**Example 8:** Consider the RBD shown in Fig. 1.15. It is required to compute the overall reliability of the system in terms of the reliability of its individual components using the Bay's theorem. In the given RBD, we can choose module 5 as module A and redraw the RBD under the two conditions

1. The system with A working
2. The system with A not working

The reliability of each of the resulted sub-systems can now be computed as follows.

$$R_{A \text{ is OK}} = [1-(1-R1)(1-R2)][1-(1-R3)(1-R4)]R5$$

$$R_{A \text{ is Faulty}} = [1-(1-R1*R3)(1-R2*R4)](1-R5)$$

One of the main versatility of Bay's theorem is that it can be applied recursively to each of the sub systems, if needed be.

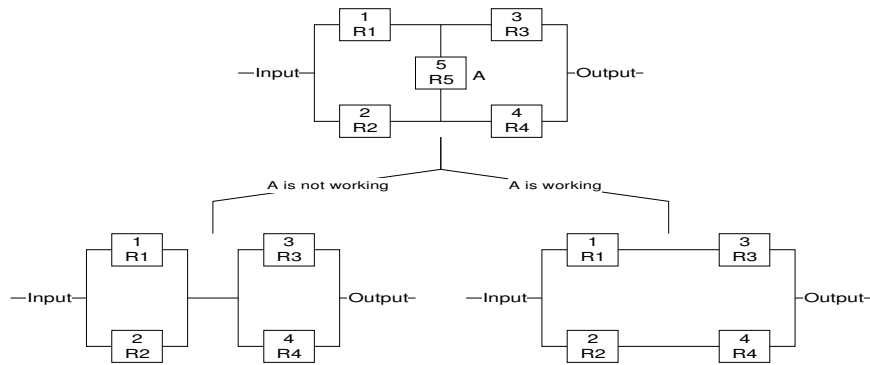


Fig. 1.15: Example RBD for applying Bay's theorem.

### An approximate model

The approximate model provides an upper bound on the reliability of a given system in terms of the reliability of its components. The model assumes that the system consists of parallel paths from input to output and computes the reliability using the parallel model. Consider, for example, the reliability block diagram (RBD) shown in Fig. 1.15. The approximate (upper bound) reliability between the input and the output of the system represented by this RBD can be computed assuming that there exists four (parallel) paths as follows:

- Path #1: consisting of components 1 and 3.
- Path #2: consisting of components 1, 5, and 4.
- Path #3: consisting of components 2 and 4.
- Path #4: consisting of components 2, 5, and 3.

Therefore, the approximate (upper bound) reliability between the input and the output of the system is computed as

$$R \leq 1 - \prod_{i=1}^4 (1 - R_{path_i}) \leq 1 - (1 - R_1 R_3)(1 - R_1 R_5 R_4)(1 - R_2 R_4)(1 - R_2 R_5 R_3)$$

Although simple and straightforward to use, the combinatorial model has a number of limitations. Among these, is the inability to model things such as system repair and availability. In addition, the combinatorial model always leads to complex reliability expressions. An alternate model is the discrete (Markov) model.

## (2) Discrete (Markov) Model

In simple terms, the discrete (Markov) model is based on the concept of state and state transition. These are recorded during repeated discrete time slices,  $\Delta t$ . System state is a combination of faulty and fault-free states of its components. State transition shows the changes in the system's state as time progresses. Consider, for example, the case of a system consisting of a single module (component), as shown in Fig. 1.16. At any given time, the module can exist in one of two states, i.e., Fault-free or Faulty. We can draw a state diagram for that module as follows.

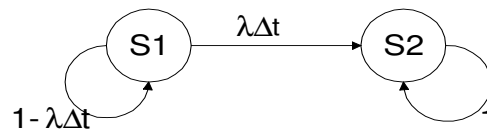


Fig. 1.16: A single component system states with no repair.

In that state diagram, circles represent states and arcs represent state transitions. State S1 represents the fault-free state while state S2 represents the faulty state. The failure rate of the module is assumed to be  $\lambda$ . The versatility of the discrete model becomes clear if we assume the case of having repair (with repair rate  $\mu$ ) for the same single module system. In this case, the state transition diagram will become as shown in Fig. 1.17.

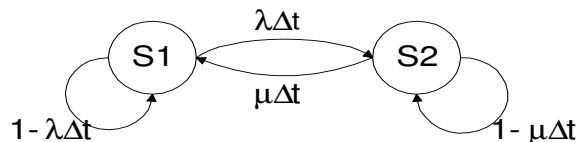


Fig. 1.17: A single component system states with repair.

It is now possible to write probability equations relating the state of the system at a given time  $t$  and that at a subsequent time  $t + \Delta t$ . These are shown below.

$$P_1(t+\Delta t) = (1-\lambda\Delta t)*P_1(t)+\mu\Delta t*P_2(t)$$

$$P_2(t+\Delta t) = \lambda\Delta t*P_1(t)+(1-\mu\Delta t)*P_2(t)$$

Writing this in a matrix form will result the following.

$$\begin{bmatrix} P_1(t + \Delta t) \\ P_2(t + \Delta t) \end{bmatrix} = \begin{bmatrix} 1 - \lambda\Delta t & \mu\Delta t \\ \lambda\Delta t & 1 - \mu\Delta t \end{bmatrix} \begin{bmatrix} P_1(t) \\ P_2(t) \end{bmatrix}$$

or simply  $\underline{P}(t+\Delta t) = \underline{A} * \underline{P}(t)$ , where  $\underline{A}$  is called the state transition matrix. This matrix form can be used to compute the probability of the system existing in a given state at any time. For example,  $\underline{P}(0+\Delta t) = \underline{A} * \underline{P}(0)$ , where  $\underline{P}(0)$  is the initial state of the system. Similarly,  $\underline{P}(2\Delta t) = \underline{A} * \underline{P}(\Delta t) = \underline{A} * \underline{A} * \underline{P}(0) = \underline{A}^2 * \underline{P}(0)$ . In general  $\underline{P}(n\Delta t) = \Delta t) = \underline{A}^n * \underline{P}(0)$ .

**Example**

Consider the case of a TMR. The state diagram is shown in Fig. 1.18 (assuming that  $X = \lambda\Delta t$ ).

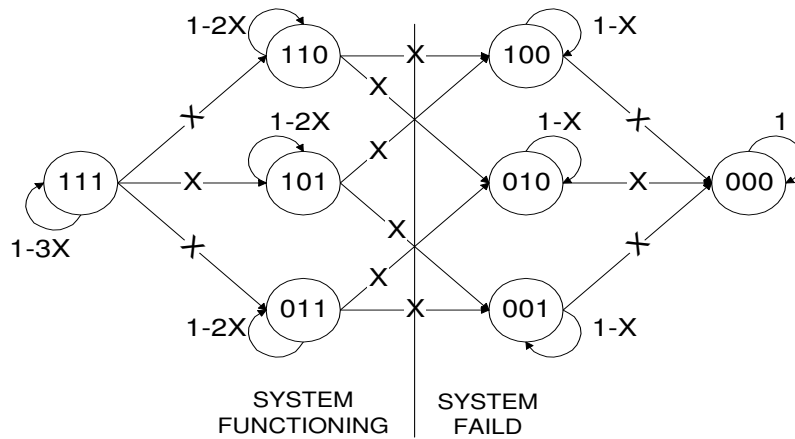


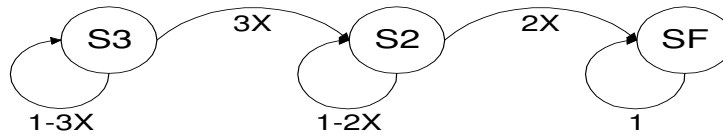
Fig. 1.18: TMR state diagram.

This can be reduced to 3 states as follows:

S3: State whereby three modules are working

S2: State whereby two module are working

SF: State whereby 1 or 0 module is working (Failure state)



Writing the probability equation in the matrix form will result in the following.

$$\begin{bmatrix} P_3(t + \Delta t) \\ P_2(t + \Delta t) \\ P_F(t + \Delta t) \end{bmatrix} = \begin{bmatrix} 1-3X & 0 & 0 \\ 3X & 1-2X & 0 \\ 0 & 2X & 1 \end{bmatrix} \begin{bmatrix} P_3(t) \\ P_2(t) \\ P_F(t) \end{bmatrix} \text{ and } \underline{P}(0) = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

It should be noted that the shorter the  $\Delta t$  period, the accurate the model. The case whereby  $\Delta t = 0$  leads to the continuous model. Consider, for example the case of the TMR. The following mathematical treatment shows how to obtain the reliability equations starting from those obtain from the discrete model.

$$P_3(t+\Delta t) = (1-3\lambda\Delta t)P_3(t)$$

$$\lim_{\Delta t \rightarrow 0} \frac{P_3(t + \Delta t) - P_3(t)}{\Delta t} = \frac{dp_3(t)}{dt} = -3\lambda p_3(t)$$

$$P_2(t+\Delta t) = (3\lambda\Delta t)P_3(t) + (1-2\lambda\Delta t)P_2(t)$$

$$\lim_{\Delta t \rightarrow 0} \frac{P_2(t + \Delta t) - P_2(t)}{\Delta t} = \frac{dp_2(t)}{dt} = 3\lambda p_3(t) - 2\lambda p_2(t)$$

$$P_F(t+\Delta t) = (2\lambda\Delta t)P_2(t) + P_F(t)$$

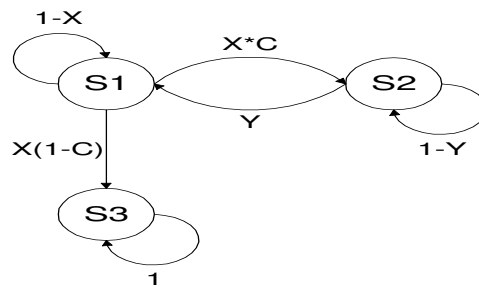
$$\lim_{\Delta t \rightarrow 0} \frac{P_F(t + \Delta t) - P_F(t)}{\Delta t} = \frac{dp_F(t)}{dt} = 2\lambda p_2(t)$$

The above three simultaneous partial differential equations can be solved using *Laplace* and the inverse *Laplace* transforms as follows.

$$\begin{aligned}
 S \cdot P_3(S) - P_3(0) &= -3\lambda \cdot P_3(S), \text{ i.e., } P_3(S) = 1/(S+3\lambda) \text{ and } P_3(t) = e^{-3\lambda t} \\
 S \cdot P_2(S) - P_2(0) &= 3\lambda \cdot P_3(S) - 2\lambda \cdot P_2(S), \text{ i.e., } P_2(t) = 3e^{-2\lambda t} - 3e^{-3\lambda t} \\
 S \cdot P_F(S) - P_F(0) &= 2\lambda \cdot P_2(S), \text{ i.e., } P_F(t) = 1 - 3e^{-2\lambda t} + 2e^{-3\lambda t} \\
 P_{TMR} &= 1 - P_F = 1 - (1 - 3e^{-2\lambda t} + 2e^{-3\lambda t}) = 3e^{-2\lambda t} - 2e^{-3\lambda t}
 \end{aligned}$$

This same result would have been obtained using the continuous model.

The versatility of the discrete model can be displayed by considering the effect of factors such as the fault coverage (FC), *C*. The FC is defined as the ratio of the number of faults that a diagnostic system can detect to the total number of possible faults occurring in a given module. The above discussion assumes that *C* = 1. However, if *C* < 1 then there will be some more states to be considered in the state transition diagram. These states correspond to the case(s) whereby the fault has occurred but was not detected. Consider, for example, the single module system with fault coverage *C* < 1. The new state transition diagram will become as shown below.



$$X = \lambda \Delta t, Y = \mu \Delta t$$

S1: The system is working

S2: The system is faulty, and the fault was detected by the diagnostic system

S3: The system is faulty, and the fault was not detected by the diagnostic system

The *availability* of the system can then be computed as the probability of being in  $S1 = P_1(t)$ . The *safety* of the system is the probability of being in  $S1$  or  $S2 = P_1(t)+P_2(t)$ . The un-safety of the system is the probability of being in  $S3 = P_3(t)$ .

#### 1.4 Summary

The main objective of this chapter has been to introduce the reader to the some of the fundamental concepts in fault tolerance and reliability analysis. In fulfilling this objective, we have introduced the main issues related to the design and analysis of fault-tolerant systems. We have also discussed different types of faults and their characterization. A detailed discussion on redundancy and its basic forms, i.e., hardware, software, time, and information has been conducted with illustrative examples. In addition, we have covered the fundamentals of reliability modeling and evaluation techniques. These include the combinatorial and the discrete (Markov) reliability modeling.

#### References

##### *Books*

- [1] Pradhan, D. (Editor), Fault-Tolerant Computer System Design, Prentice-Hall PTR, New Jersey, 1996.
- [2] Pradhan, D. and Avresky, D. (Editors), Fault-Tolerant Parallel and Distributed Systems, Computer Society Press, Los Alamitos, California, 1995.
- [3] Jalote, P., Fault Tolerance in Distributed Systems, Prentice-Hall PTR, New Jersey, 1994.

##### *Websites*

- [1] <http://elib.uni-stuttgart.de/opus/volltexte/2000/616/pdf/Diss.pdf>
- [2] [http://www.omg.org/news/meetings/workshops/presentations/embedded-rt2002/03-1\\_Garon-Narasimhan\\_FTTutorial-OMGWkshp-2002.pdf](http://www.omg.org/news/meetings/workshops/presentations/embedded-rt2002/03-1_Garon-Narasimhan_FTTutorial-OMGWkshp-2002.pdf)

- [3] <http://www.csee.wvu.edu/~katerina/Teaching/CS-757-Fall-2003/Fault-Tolerance.pdf>
- [4] <http://www.cs.vu.nl/~ast/books/ds1/07.pdf>
- [5] <http://www.ecs.soton.ac.uk/~lavm/papers/fta-europar2002.pdf>
- [6] <http://wwwhome.cs.utwente.nl/~krol/publications-krol/proefschrift/ts1m.pdf>
- [7] <http://www.idi.ntnu.no/~noervaag/IDI-TR-6-99/IDI-TR-6-99.pdf>
- [8] [http://www.cin.ufpe.br/~prmm/wellington/00995536\\_wjs.pdf](http://www.cin.ufpe.br/~prmm/wellington/00995536_wjs.pdf)
- [9] <http://ii.pmf.ukim.edu.mk/ciit/2001/papers/2Ciit-24.pdf>