ERROR DETECTION AND CORRECTION TO IMPROVE DATA BASE RELIABILITY

by

Sylvia F. Dalton

December 1974 TR-40

This work was supported in part by Air Force Contract #F33600-74-C-0314, Data Base Validation and Integrity Management Control Study.

TABLE OF CONTENTS

Chapter		Page
I.	INTRODUCTION AND OVERVIEW.	1
	Introduction.	1
	Previous Research on Error Correction	6
II.	NEED FOR AUTOMATIC ERROR HANDLING.	9
	Sources of Computer Errors.	9
	Probability of Errors in a Data Base.	12
	Increasing Reliability in Computers	14
III.	ERROR DETECTION AND CORRECTION LOGIC	19
	Introduction.	19
	Data Base Considerations.	19
	Incorporating Data Integrity Management (DIM) Into DMS.	20
	Selecting the Packet Size	22
	Maintaining the Redundancy.	23
	Possibility of Errors in the Redundancy . . .	24
	Detection	25
	Status Indicator for a Block	26
	Initial Analysis to Detect Old Errors. . .	26
	Detecting Current Errors	26
	Final Analysis	29
	Error Correction.	30
	Initial Analysis	31

Chapter | Page

Looping Through the Reconstructor. 31

Reconstructor Logic. 32

Final Analysis 33

Validation of One Data Block 34

Storing the Changed Data 35

Interaction With the Staff Personnel. 36

Suggestions for Background Validation 37

Summary Using a Model of System Logic Including Detection and Correction. 38

IV. ALGORITHMS FOR ERROR DETECTION AND CORRECTION. . 45

Parity Schemes. 45

Limitations of Parity and Checksum. 48

Error Correction Algorithms 51

Algorithm HV 52

Algorithm HVC. 57

Algorithm HVZ. 58

Discussion of Diagonal Parities in Algorithms 60

Algorithms VD and VH 61

Algorithm VDC. 61

Algorithm HVD. 62

Summary of Error Correction Algorithms. . . . 66

V. A SAMPLE COST ANALYSIS 69

Parameters Involved in a Cost Analysis. . . . 69

The Cost of Errors Without a Detection/ Correction Procedure. 72

Chapter Page

The Cost of an Error Detection/Correction Procedure. 73

A Model to Analyze the NOLS System 74

Summary of the Cost Analysis 81

VI. SUMMARY 83

APPENDICES

A. Resolving Ambiguities for Algorithms HVC or VDC 85

B. Algorithm HVD in Detailed Logic 92

C. COMPASS Subroutine to Compute Redundancy for Algorithm HVD 96

BIBLIOGRAPHY 100

LIST OF TABLES

Table		Page
1.	Probability of Errors in the Data Base.	12
2.	Status Indicator for a Data Block	27
3.	Detection Limitations Using a Checksum.	49
4.	Error Correction Algorithms	53
5.	Checksum Comparisons.	57
6.	Summary of Error Correction Algorithms.	67
7.	Comparison of Error Correction Algorithms	68
8.	Results of the NOLS Model Analysis.	80
9.	Algorithm HVC or VDC Logic When Two Bits Turned Off.	86
10.	Algorithm HVC or VDC Logic When Two Bits Turned On	88
11.	Algorithm HVC or VDC Logic When Two Bits Reversed.	90

LIST OF FIGURES

Figure		Page
1.	Organization of First-Order and Second-Order Check Digits.	7
2.	Data Integrity Management Network Model, Introduction of Errors.	39
3.	Data Integrity Management Network Model, Processing of Errors.	40
4.	Horizontal and Vertical Parities.	46
5.	Horizontal, Vertical, and Z-axis Parities	47
6.	Left and Right Diagonal Parities.	47
7.	Left and Right Diagonal Parities With Wrap-around.	48
8.	Error Correction Using Horizontal and Vertical Parities.	54
9.	Ambiguity of Two Erroneous Bits Using Algorithm HV.	56
10.	Ambiguity of Multiple Erroneous Bits Using Algorithm HVZ	59
11.	Ambiguity of Two Erroneous Bits Using Algorithm VDC	63
12.	Error Correction Using Horizontal, Vertical, and Diagonal Parities (Left-Diagonal with Wraparound.	65
13.	Ambiguity of Two Erroneous Bits Using Algorithm HVD	65
14.	A Simplified Model of NOLS.	75

CHAPTER 1

INTRODUCTION AND OVERVIEW

INTRODUCTION

Reliability in computer data bases is becoming an increasingly important topic. Data integrity is needed for the volumes of vital statistics kept in long term storage such as disk. This data is used in computer processing to produce analysis and decisions which may affect human lives. Complex, essentially continuously running systems cannot be adequately checked by a staff of people because of the combinatorial explosion of predicting and analyzing errors. Timing constraints tend to make human control impractical or even impossible. For example, a computer application monitoring a spacecraft launching is designed to instigate an abort without human authorization.

There are many causes for the existence of erroneous or invalid data. The data base resides on a large number of storage media which can experience failures. The data must be frequently transmitted through Input/Output channels (I/O) with appreciable levels of noise. The software system supporting the data base activity can contain subtle or unknown errors which may perturb the data.

The problems of inaccurate or invalid data are intensified by the criticality of the data and by the per-

formance and output demanded of an application. Human lives or valuable processes may be jeopardized. Yet there have been virtually no systematic studies made of possible methods of error control and error recovery in data base systems.

Automation points to using the computer to improve its own reliability. Hardware capabilities should be augmented by software functions. Software techniques can provide programs to detect and correct errors--before these errors are used in further processing.

This paper is a discussion of automated error processing to improve data base reliability. Primary emphasis is placed on software techniques to detect and correct errors. Various integrity management algorithms are presented with a comparison of effectiveness and resource requirements. The algorithms basically involve introduction of some redundancy computed from the original data. Changes in the data from the original may be identified by a comparison of stored redundancy with a newly computed redundancy taken from the current data.

These algorithms are based on concepts developed in communication theory [1]. This required redundant strings of parity taken on each row and on each column (horizontal and vertical parity). This paper converts the scheme into a software environment, expanding the theory to include new algorithms based on various diagonal parity

strings. Such an amount of redundancy increases the data base size by approximately 5 percent.

A detection function can check the data by comparison with the redundant information. Thus erroneous data may be identified although the exact number or locations of the errors do not need to be pinpointed by detection. The component failure causing the problem is not identified. An error correction function must use the redundancy to locate the exact positions of errors. Then the setting of those positions may be reversed to return the data to an error-free state matching the original copy.

Error detection and correction algorithms may be categorized by the amount and type of redundancy required. Any algorithm has a predictable performance when presented with different types of errors. Any algorithm has its limitations which allow various types of self-compensating errors to camouflage each other and remain undetected. Any error correction algorithm would have classes of detected errors which can not be corrected automatically. For example, majority vote involves three copies of the same data. Majority vote throws away the unmatching copy of data when the other two copies agree. The algorithm cannot handle the case of three different copies (which means two or three copies have experienced errors).

This paper presents algorithms that are deemed

to cover a significant percentage of possible errors at a minimal cost in system resources. This paper recommends that detection use the same amount of redundancy as error correction uses. (A more limited detection algorithm should be used only if system timing is a problem.) This paper presents error correction algorithms that can automatically correct at best three erroneous bits in one unit of data. The size of the data unit can be decreased to cope with a high density of errors.

A status indicator is designed to categorize each unit of data regarding the success of error correction. The data can be marked with one of the following states: free of errors, usable although containing error, or unusable due to critical errors. Manual corrections or other updates can be used to supplement automated procedures in re-establishing an error-free data base.

Data Integrity Management (DIM) incorporates the whole concept of error handling for a data base. DIM combines the concepts of error detection and correction with the backup capabilities of manual corrections and Rollback/Restart/Recovery [2]. DIM can include validation of a block which has just been processed by error correction. DIM processing can include validation techniques before each use of the data. For example, perform suitability checks on parameter values (for proper type, range, or context with other parameters). Also check every

redundancy coded into the system such as forward and backward addresses when this is within the scope of the current processing. DIM can include background validators which run with low priority to check for errors, possibly as a continuous process. A team of people is needed to supplement computer capabilities.

Many data base systems are controlled by a Data Management System (DMS). This provides an interface between the users and the data. Many of the functions of DIM can be incorporated into the DMS with minimal impact on the users. It would be possible for the DMS to perform different degrees of detection and correction tailored for the criticality of each data table within the data bases. Limited capabilities in some phases might solve timing problems.

Costs are incurred to incorporate error detection and correction into a system. These costs must be weighed against the cost of errors in a system without such reliability. This paper presents factors to be considered in a cost analysis when deciding what degree of error processing to include in a system.

A sample model study was done to aid in the cost analysis. This estimates the effect on system response time to add error detection and correction. The system chosen for the model is the Advanced Logistics System (ALS) of the Air Force Logistics Command [3]. A subsystem of

ALS was chosen for primary emphasis because of the criticality of the data base: Nuclear Ordnance Logistics System (NOLS).

PREVIOUS RESEARCH ON ERROR CORRECTION

Research on error correcting techniques has been done in several related fields: communication theory, information theory, fault-tolerant computing, and automata theory.

Communication theory is concerned with the transfer of data over teletype or telephone lines which can degrade with noise. Special equipment could be provided with special circuity for each end of the communication line to increase reliability: an encoder to recode the data with redundancy and a decoder to reformat the data with error correction as needed.

The list of publications in this area is lengthy but often not directly applicable to this paper. Of historical interest is a paper by Elias [1] describing iteration of simple error correcting codes developed by Hamming [4]. This code provides single-error-correction and double-error-detection (SEC-DED). There is the drawback that any odd number of erroneous bits would be corrected as if only one erroneous bit had been found. This in effect introduces an extra error. Based on [1], Figure 1 shows an $N_R x N_C$ rectangular array with redundancy increasing the block to size $(N_R+C_R)x(N_C+C_C)$. The checks C_R on

each row and checks C_C on each column are termed first-order checks. The checks on checks are called second-order checks; these are the same whether computed from an error-free C_R or C_C.

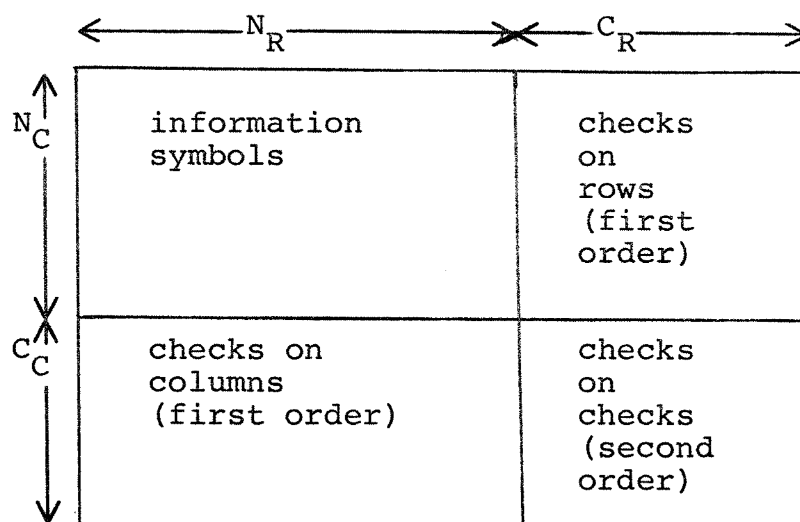

Figure 1: Organization of First-Order and Second-Order Check Digits

The length of C_R is dependent upon the length of N_R. For example, if N_R is 60 bits, then C_R must be at least 8 bits.

As a general rule, communications theory is not overly concerned with minimizing the amount of redundant data. Specialized encoders and decoders handle the redundancy. Therefore polynomial codes such as Hamming [4] are preferred to a rectangular parity scheme involving much less redundancy [5].

Fault-tolerant computing is defined by Ramamoorthy [6] as "the ability to execute specified algorithms correct-

ly regardless of hardware failures and software errors." He goes on to state that this implies the design of computers which are self-diagnosable and self-repairing.

Two examples of hardware reliability schemes are referenced here because of their direct correlation with this thesis. Patel and Hsiao [7] present an error correction technique to be used for a computer memory system. It is an extension of Hamming codes [4] which requires more than 8 bits of redundancy per row of 64 bits. As another example, Peterson [8] describes a scheme of parity (1 bit or redundancy) taken on each row and each column with one second-order check bit. Peterson states that this binary iterated code is actually used for error detection on magnetic tape units used by IBM computers.

To summarize, research into the type of error correction presented in this thesis has been published for over twenty years. It generally involved communication networks with specialized equipment. The scheme was used in some computer hardware development. It was not particularly presented as a technique to be incorporated into software.

Parity taken on each row and each column within an array of data is a generally discussed method [5, 7, 8, 9]. However, reference to a diagonal parity was not found in the literature available to the author. No more elaborate schemes involving parity were found.

CHAPTER II

NEED FOR AUTOMATIC ERROR HANDLING

SOURCES OF COMPUTER ERRORS

This paper addresses the smallest computer error: a change in the setting of one bit. A bit is a physical entity which can take on only the values of 0 or 1. If the setting is sufficient to register as present, then the bit is said to be "on" or set to a "1". If the "on" condition is absent (or below a certain threshold) then the bit is termed "off" or set to "0".

Groups of bits make up computer storage where information or data can be placed. There are several physical representations of bits used to make up computer storage; for example, magnetic core or magnetic tape. While a bit is expected to keep its current setting until altered in the flow of computer logic, possibly the bit could change accidentally. Such an error could be caused by either hardware or software failure.

Hardware failure could be caused by a variety of problems. Unusual environmental occurrences such as electrical surges or temperature variations could affect any bit. On the other hand, preventive maintenance might not have caught a bit which is beginning to fail due to wear. Such a bit could become sensitive to the settings

of neighboring bits and change to match. In some computers a bit is not refreshed if it is not accessed. Thus a seldom used bit could age; that is, an "on" setting could slowly dissipate until it goes below the threshold to be considered a "1".

Input/Output devices transfer information between various storage devices. Such hardware can introduce errors into the settings of the bits. Thus the newly moved copy does not exactly match the original copy.

Software failures are caused by faulty design or implementation of programs. These problems appear primarily in the initial stages of testing a program and upon integration into a package with other programs.

In an attempt to classify errors, Martin [10] provides three categories:

1. A solid or permanent error which repeatedly occurs whenever the fault is encountered.
2. An intermittent error which occurs only occasionally and is difficult to reproduce. It is usually caused by unfavorable environmental tolerance conditions.
3. A transient error which occurs only once and cannot be made to repeat itself. It is caused by such things as noise or a brief abnormal condition in a supply line.

This paper is directed towards storage problems

involving intermittent or transient errors which are characterized by a finite number of spurious bit changes. The proposed methods are most effective if the erroneous bits are not too densely packed. Note that this class of errors normally does not automatically register as a hardware or software failure. Therefore special techniques are required even to find that an error has occurred. The original hardware or software problem which caused the incorrect bits is immaterial; the problem is not identified.

This paper is directed more towards hardware failures than software failures for two reasons. First, "it is generally agreed that once a software package is completely debugged then it is 100 percent reliable and, unlike hardware, does not deteriorate in time." [11] Secondly, a software error is more likely to be totally destructive to any data that it perturbs.

It is assumed that implementation of these methods will yield correct software and that the system software will function without error while this implementation is executing.

There are a multitude of other computer problems. Many are of a catastrophic nature which require hardware repair or software alteration. For example, a disk head crash may destroy some data. This paper does not address such problems although some of the proposed methods may possibly be useful.

PROBABILITY OF ERRORS IN A DATA BASE

Errors are unpredictable accidents by the definition of computer errors as used in this thesis. Thus it becomes extremely difficult to quantify the number of errors which can be expected or to predict reliability in general terms.

Discussions of reliability become most meaningful for specific cases. Hardware manufacturers should have some statistics on predicted reliability of a component. System analysts could make some predictions about a grouping of components found in a particular hardware configuration. The best source of statistics involving reliability would come from measurements taken on a running system.

Noting the problem of generally discussing reliability, Martin [5] provided some numbers merely as examples. (See reference 5, Table 2.1 on page 12 for more details.) These estimates are summarized in Table 1 showing the probabilities of errors in a data base.

Probability of Errors:	
hardware/software error damages file-	
loss of entire file-	might happen once in 40 years
loss of single records-	might happen once in 100 years
modification of records-	might happen once in 10 days
data transmission error not detected-	
loss of single records-	might happen once in 100 days
modification of records-	might happen once a day

Table 1: Probability of Errors in the Data Base

These numbers do support efforts which focus on problems with modification of records. Modification of a record occurs 10 to 100 times more often than loss of a single record.

Discussing the modification of records leads to a consideration of the extent of modification. Techniques to handle errors are often concerned with exactly how many bits are erroneous and with their density. Although failure rates can be correlated to the hardware and software composing a particular system, a general rough estimate can be made. Gear [9] states, "If the probability of a bit error in a word is independent of all other bit errors, and is, say 1 chance in 10^9 for each memory fetch, then the probability of 2-bit errors simultaneously is 1 chance in 10^{18}." Continuing in this manner, according to probability theory the probability of n-bit errors simultaneously is 1 chance in 10^{9n}.

For guidelines, this author would generalize the occurrence of errors with emphasis on the physical proximity of bits. Considering all of disk storage, errors in physically disjoint areas of storage are considered to be independent of each other and thus follow traditional probability theory. However, within physical proximity of each other, bits are not always independent. For example, a physical unit of storage may be experiencing hardware fatigue due to wear. All bits within that

unit are more likely to fail. As another example, when an Input/Output device fails, multiple bits in the record may be altered.

For guidelines in this paper, the following generalizations are presented. The probability of a 1-bit error occurring in a data base is 1 chance in 10^8 for each microsecond. The probability of two physically disjoint 1-bit errors is 1 chance in 10^{16} for each microsecond; continuing with the probability that n physically disjoint bit errors is 1 chance in 10^{8n} microseconds. The probability of two physically close 1-bit errors (called a 2-bit error) is 1 chance in 10^{11} for each microsecond; continuing with the probability that n physically close bit errors is 1 chance in $10^{8+3(n-1)}$ microseconds.

INCREASING RELIABILITY IN COMPUTERS

Hardware design is improving reliability. Components have a longer mean time between failures. Also error handling features are sometimes designed into the hardware. In addition, software techniques are increasing reliability. For example, system functions handle critical areas, providing protection against careless users. Program correctness (formal software verification) is a current topic of considerable research and development work.

Computer errors still occur although reliability is improving. The spectrum of computer problems ranges from an improper setting of a single bit to a catastrophic

complete halt in processing. The smallest error can precipitate further errors in the computer processing. For example, an incorrect address may point to meaningless information. Such fallacies could further degrade the integrity of the system until processing might completely halt.

Reliability is of grave concern where time is an important consideration. Real time systems can suffer serious delays caused by errors. Interactive systems can jeopardize performance and user confidence by incorrect responses or blackouts of service. Reliability is an economic concern when the computer controls valuable processes.

With increasingly complex, centralized computer applications which may run continuously for years, the problem of reliability is greatly intensified. A change in one bit can be an insidious error. Such applications are generally characterized by tremendously large amounts of data, too large to record on hard copy periodically. Even a team of people assigned to maintain the data cannot verify that it is correct; they must generally serve as "fire fighters" to solve the most pressing problems. Any chance for compromised data integrity undermines user confidence even when the data is error-free.

Automation points to further utilization of the computer--directing it in ways to improve its own reliability. Each time a piece of data is used, the computer

could examine it for errors in a detection function. If errors have occurred which are relatively simple to diagnose, the computer could automatically correct the data in a correction function.

Automatic error correction is not designed to repair every degree of error. Some cases are ambiguous and cannot be deterministically (100 percent accurately) corrected algorithmically. Some cases involve too great a degree of data change. For example, the original data may have been accidentally destroyed by an overwrite. Backup to automatic error correction must be provided. The computer could notify the team of people to prepare corrections to input. If the system is seriously degraded, the computer could be restarted after it is selectively refurbished with histories of data. Hopefully this technique of rollback would not re-introduce problems [2].

Installations may have to devote a significant percentage of total computer resources for error handling. Extra storage is needed to save some redundant information. Computer processing time, central memory while executing, and Input/Output devices are needed. The cost can be regarded as insurance, balanced against the small percentage of errors that do occur and are prevented from causing further damage.

In a system without adequate error handling, an error which is not identified may have a high cost. This

primarily depends on the type of data in which the error occurred. A data base is composed of two levels: a data structure which defines the positional meaning of the data and the parameter values which make up the data. The structure is extremely important to system integrity. An invalid or incorrect address used to access other information allows the possibilities of system failure or error propagation within the data base. Addresses or indexes are particularly prone to start a "domino effect" which could lead to a complete halt of processing. Incorrect data values allow the possibilities of many errors such as incorrect reports, mishandling of inventory, or error propagation within the data base. Data could describe critical information such as the location and range of a guided missile.

Another high cost may be associated with an error which is identified but not corrected immediately. The cost is determined by the significance of the data. The information may be needed to complete an interactive request or to proceed with a lengthy report. The computer measures delays in units of nanoseconds. Many real time systems and interactive systems cannot tolerate time delays. It is estimated that, on the average, a correction entered by a team of people would require an hour of research. If errors were occurring at a faster rate than they could be corrected by people, a bottleneck could build up until many needed

functions of computer activity would be degraded.

Time delays would be minimized if the computer software was designed to correct errors as soon as possible. Bottlenecking would be minimized.

Management decisions must weigh the cost of errors against the cost of error processing. These may involve intangibles. Practicality moderates the selection of error handling methods. For a heavily loaded system with timing restraints, error handling might be allowed only in slack periods. For a system with storage shortages limiting redundant data, a suitability check is still possible. For example, make checks on maximum/minimum values or context with other parameters.

To summarize, Senko et al. [12] state: "Advanced computerized information systems support their primary functions by secondary functions such as security, recovery, input checking, and scheduling of system resources."

CHAPTER III

ERROR DETECTION AND CORRECTION LOGIC

INTRODUCTION

A general description of error detection and correction are presented in this chapter. Detailed algorithms to be used by these functions are presented in Chapter IV.

The first part of this chapter is devoted to the relationship between the data base and DIM. The size of the data unit to be presented to error handling at one time is discussed. Storage of the redundancy is considered. The possibility of errors in the redundancy is analyzed.

Error detection logic is presented covering both old and new errors. Error correction logic is detailed. Interaction with the staff is outlined. Recommendations for background validation are made.

A summary of this whole chapter presents a model which shows system logic including error detection and correction.

DATA BASE CONSIDERATIONS

A data base is typically composed of a variety of data tables. Each data table could have a unique definition of structure for addresses and parameter values. Each data table could have an organization where corres-

ponding strings of information are in the same hierarchical level. Each string may be stored in disjoint areas on the disk. A string could be composed of substrings which may be physically non-contiguous with forward and backward addresses stored in each substring.

All the logical strings composing a data table may be indexed in another data table. In fact, pyramidal indexing scheme may be employed to speed access of a particular block. This could involve multiple data tables of indexes.

The need for DMS (Data Management System) becomes more apparent as the data base system becomes more complicated with various data tables and indexing schemes. DMS must perform a mapping function between the logical string and physical storage. DMS centralizes the logic of data table structure and parameter values. It frees the user from most considerations of data management. The DMS also provides a buffer to protect the data base from careless users.

INCORPORATING DATA INTEGRITY MANAGEMENT (DIM) INTO DMS

The proposed DIM functions would best be incorporated into the DMS. Error detection and correction are logical concepts that are performed on one logical string or substring of data at a time. The actual mechanics of the process must work at the physical level. This discussion will define a <u>block</u> to be a logical unit of data, not

necessarily stored contiguously, which must be presented to the error handling functions at one time as the data is readied for use.

Error correction is actually performed by repeated application of an algorithm (in a looping process). For clarity, define that the reconstructor represents one execution of the algorithm. The algorithm is designed to handle one rectangular array of bits, termed a packet, with its corresponding redundancy. A packet is an nxm matrix of physically contiguous bits. The dimensions of the packet are the same for all data and remain fixed (see discussion below in SELECTING THE PACKET SIZE).

Error detection can be implemented in a scheme closely paralleling that used for error correction. One execution of the algorithm would take place in the detector. The detector would accept as input one packet of fixed dimensions with its corresponding redundancy. (Detection may be implemented alternatively on the block level with no breakdown into packets.)

Consider the total process of breaking a block into physical packets of fixed dimension, assuming that the block size is larger than one packet size. DMS must handle the retrieval of the string (or a part of it) from disk into central memory. The block must be presented to detection and possibly to error correction to ready it for use. This requires DMS to perform a mapping function between physical storage and packets. This would be done in a looping

process by presenting one packet at a time to the reconstructor with its corresponding redundancy. Any unit of data smaller than a packet must be temporarily padded with zeroes. (This would not change the redundancy if even parity is used.)

SELECTING THE PACKET SIZE

A packet is an nxm matrix of contiguously stored bits. The sizes for n and m would be selected by system designers. Hardware specifications and system data structure would control this decision. The optimal size for both n and m would be a word width w in central memory. This would be very efficient for describing all of disk storage since the total size of the redundancy would be minimized. The algorithm could function very efficiently with n=m=w. The algorithm would be complicated if the column width were longer than the word width, necessitating multiple words to store vertical or diagonal parities. If the amount of data readied for use at one time had a length less than w, then m<w would be best for the algorithm but would require more total storage to maintain the redundancy. A problem with density of errors might motivate decreasing m for a shorter packet.

It is recommended that n and m remain fixed once selected. The algorithm could operate more efficiently without an input of variable dimensions. Storage of variable dimensions with the redundancy would increase storage

requirements significantly. The DMS can temporarily pad with zeroes any unit of data smaller than size m. This would not change the redundancy if even parity is used.

Control Data Corporation equipment has a word width of 60 bits. This provides a natural packet size 60 words in length. Thus the nxm matrix size would be 60x60 bits for a total of 3600 bits per packet. Estimates in this paper are sometimes based on this figure (with proper clarification given at each estimate).

MAINTAINING THE REDUNDANCY

The redundancy required for the data base values may be stored with the data it describes or separately. Storage with the data necessitates a slighly longer total unit of data (approximately 5 percent increase in length). This would have little or no effect on the Input/Output timing. Storage separate from the data seems to have a higher cost than storage with the data. It requires an extra Input/Output command to store or retrieve the redundancy. This separate storage must be correlated in some way to the data. One of the following methods may be chosen:

- An address in the data pointing to the position of its redundancy on the disk.
- An address in the indexing scheme associating the data with its redundancy.

- An algorithmic scheme implicitly correlating the position of the data on the disk with a position of the redundancy within an area devoted to storing all the redundancy for that disk.

POSSIBILITY OF ERRORS IN THE REDUNDANCY

Hardware parity schemes can protect the parity information. For example, the parity is not brought over the Input/Output channels. Software schemes cannot provide any special protection for the redundancy. This information is subject to error in exactly the same manner as the data it describes. Thus storage problems, I/O failures, and software inaccuracies can harm the redundancy.

Second-order checks may be implemented to detect errors within the redundant information. This is diagrammed in Figure 1. These checks may be implemented as a checksum, or as one or more bits of parity. This would require additional storage and processing time. It might require additional I/O time.

This paper recommends a consideration of second-order checks when analyzing a specific system. The cost of implementing second-order checks must be weighed against the costs of errors in the redundancy.

There are several reasons that second-order checks might not be included in DIM. The amount of redundancy is very small compared to the data and thus much less

subject to error on a percentage basis. This means that an error occurs in the redundancy rather than in the data less than 5 percent of the time. These errors will be detected by noting a change in the redundancy. Detection will operate with the same success as if the error occurred in the data. If automatic error correction is performed, validation of the block may reject it immediately as unusable. If automatic error correction is not possible, backup measures can analyze the problem and cause the redundancy to be recomputed.

This paper predicts that invalid automatic error correction due to errors in the redundancy will occur for approximately 1 percent of all detected errors if second-order checks are not implemented.

The detailed algorithms presented in Chapter IV do not include any second-order checks.

DETECTION

Detection is the process of determining whether errors have occurred in the data base. Detection is performed on a portion of data as it is being readied for use. The exact location of any errors in the data is not needed.

Detection must exist on two levels. First, an old error may exist in the block which could not be automatically corrected and has not received additional attention. Secondly, a new error may have occurred since the data was last readied for use.

Status Indicator for a Block

Old errors which have not been corrected should be marked by a status indicator for each block. Codes should be in patterns that will increase the probability of recognizing an erroneous code. For example, consider the use of five bits for the status indicator as shown in Table 2.

Note that the status indicator is created to stop a sequence of re-investigating the same errors in a block every time it is to be used. Error analysis should occur only on the first detection of an incorrect block. If successful error correction is not possible, it could be a matter of hours or even days before corrections are available. During that time, the incorrect block is not continually re-analyzed.

Initial Analysis to Detect Old Errors

The detection function should first look for a status indicator of "unusable" marking a block. An indication of "unusable" makes the presence of new errors in the block immaterial. No further detection need be performed. The function should proceed to final analysis with an output parameter of "unusable".

Detecting Current Errors

Many different detection methods may be devised to detect new errors. Detection can be implemented on

CODE	MEANING	EXPLANATION	ACTION RECOMMENDED
00000	correct	correct block when last checked	detect any current error, otherwise use as needed
11111	unusable	incorrect block previously detected, errors could not be corrected so far. Either the structure of the block is bad or data integrity is critical to the system.	do not use this block
10101	usable with errors (optional feature)	incorrect block previously detected, errors could not be corrected so far. However, data integrity in not critical to the system and the structure of the block is still usable.	use this block. Note some consideration should be given to a scheme to flag as questionable all parameters used from the bad block. For example, an asterisk printed beside the data value would remind the user to question the data value.
any other code	problems, investigate	somehow the code has developed an error--a very bad sign	go to error analysis to decide usability of block

Table 2: Status Indicator for a Data Block

a block level or on a packet level. The block level would be selected where processing time and/or storage are more significant problems than the criticality of the data. (Storage is a consideration for added redundancy only if error correction is not implemented.) For example, a checksum could be packed into the header information for a block whenever the block is created or updated. A checksum is the count of the number of bits which are turned on in a block (with care to include or exclude consistently the bits of the checksum). Whenever a block is requested for use during processing, a checksum is recomputed and compared with that stored within the block. If there is no difference, the function should proceed the final analysis with an output indication of "correct". If there is any discrepancy between checksums, an error has been detected. The function should go to final analysis with an output parameter stating "needs correction".

Detection may be implemented on a packet level if the criticality of the data and other DIM concepts are considered more significant than processing time and/or storage problems. (Storage is a consideration for added redundancy only if error correction is not implemented.) This would involve looping through the detector with each packet until the whole block is checked. When an error is first detected, the looping process can cease. The function should proceed to final analysis with an output para-

meter stating "needs correction". If no errors are detected in the whole block, then the function may go to final analysis with an output indication of "correct".

Final Analysis

The detection function has produced a output parameter of "correct", "needs correction", or "unusable". The data block should be unchanged in any case.

If the output parameter is "needs correction", the error correction function should be called. This may be done optionally by bringing the logic described in the initial analysis of error correction up into this section. (In this case, the status of the block would be changed from "usable with errors" to "unusable".)

If the output parameter is "unusable", then certain users still would need to work with the block. For example, an update routine might be processing corrections entered by the staff. Other users might skip the unusable block and proceed with other processing. It is suggested that users check the status of a block before they begin using it. This seems preferable to a DMS decision of whether to halt processing in a user.

If the output parameter is "correct", no old or new errors exist in the block. This data may be used immediately by all users.

ERROR CORRECTION

Error correction is the process of using redundancy to correct errors occurring in the data base. This function would be performed by computer software as soon as possible after the error is detected.

Error correction as presented in this paper is limited to success on slightly erroneous data. Most often it is envisioned that within a finite segment of storage (corresponding to a packet), only one bit will become erroneous at any instant in time. Error correction is primarily designed for this case. Any algorithm chosen should be deterministic (completely predictable success) for a one bit error. Correction of more than one bit in error in a finite segment of storage may be indeterminant (for example, a predictable success 75 percent of the time). In the extreme, with a large percentage of erroneous bits, reconstruction would be a farce.

Error correction is presented here as a means to correct errors, not to randomly pertubate the data in a probablistic hope of improvement. Therefore any indeterminate change should not be made. Backup measures must be used.

Error correction is composed of several steps which are explained in the following sections:

- initial analysis of the error
- looping through the reconstructor until all

packets have been processed, recording each status output by the reconstructor

- final analysis of the status of the error correction process
- validation of the reconstructed data
- storing the changed data

<u>Initial Analysis</u>

Initial analysis would determine the condition of the block. The block is eligible for error correction if it contains no previous errors upon which correction failed. If the block had previously been marked with a status indication of "usable with errors", the block is indicating such a poor record of performance that it seems best just to mark it arbitrarily now as "unusable". No attempt at error correction should be made. This would end any insidious chain of invalid correction (which did appear valid until detailed usage). This measure would at times prevent valid correction to occur on a further degraded block. This logic in the initial analysis may readily be moved into the function which calls error correction, making the call contingent upon having a block with only currently detected errors.

<u>Looping Through the Reconstructor</u>

Error correction would be performed in a looping process repeatedly using the reconstructor until the whole

block is examined. Each execution of the reconstructor represents one application of the error correction algorithm on an nxm matrix of data called a packet. This looping process is based on the assumption that one block is larger than an nxm matrix.

After each execution, the reconstructor returns an indication of "success", "failure", or "no change needed". This status may be accumulated without summation (using a Boolean difference commonly called an "exclusive OR"). For example, after the block is completely processed, "failure" is just as serious whether it occurred one or three times in the looping process.

Reconstructor Logic

Various error correction algorithms are presented in Chapter IV. The exact one chosen will depend on the primary goals of an installation tempered with processing and storage considerations. The reconstructor does have some basic logic which is independent of the particular algorithm chosen.

The reconstructor should receive as input a pointer to the packet and a pointer to the corresponding redundancy. After analysis, the reconstructor can determine whether correction is needed and, if needed, whether automatic error correction is possible. If it is needed and possible, the packet is altered and an output parameter of "success" is returned upon exit. If correction

is needed but not possible, the packet is not altered and an output parameter of "failure" is returned upon exit. If correction is not needed because no errors are detected in this portion of the block, then an output parameter of "no change needed" is returned upon exit. In any case, the packet is not moved and the redundancy is unchanged.

Final Analysis

Looping through the reconstructor has built up a status indicating the success of the total error correction effort. If this status indicates that in the total process at least one change was made in the data with no failures then the data has been properly corrected. In other words, successful reconstruction indicates that at least one problem was corrected in the block and that the original redundancy now is verified.

The block may optionally be examined by validation routines before it is released for general use. Validation provides a safety check on the error correction algorithm. Approximately 1 percent of the time, an improper correction may be made if second-order checks are not implemented. Validation could catch this problem and keep the incorrect block from being used.

The status indication could show that on at least one loop through the reconstructor, an error could not be successfully corrected. The procedure to follow in this case is open to debate. Failure to reconstruct

could arbitrarily cause the block to be marked "unusable". However, an optional approach is recommended to facilitate using the correct data in a block where the errors are not detrimental to system performance. Analysis is needed to determine the significance of the errors. The block could be used if the errors are determined to be not in the structure but in the data portion of a block where the data is non-critical. The decision of criticality would probably be based on an examination of the data table name. Criticality of each data table would be indicated in a tabular definition. Thus the block could be marked "unusable" if the data is critical. If the data is non-critical, the block could be marked "usable with errors" and examined for structural validity by the validation function before it is released for use.

If marked "usable with errors", the redundancy should be recomputed for the block. This will subsequently detect an invalid reconstruction or detect further degradation in a reconstructed block.

Validation of One Data Block

Validation is the task of checking the integrity of a block to determine whether a block is usable or unusable. Validation does not state whether the block is free of errors but only whether the block is usable. The first failure to validate should end the process of checking and set the status indicator within the block to

"unusable". If no failure to validate occurs, the status indicator should be unchanged.

Validation is concerned with the two levels contained in a data table: the structure and the data values. Most important, the structural portion must be intact. Addresses and indexes should be within the proper bounds. Any redundancy coded into the data base may be checked (by performing accesses of other data blocks if necessary). For example, forward and backward pointers may be compared. Validation must examine the data within the block for suitability of type, range, or context with other parameters.

To summarize, validation is required for a block temporarily marked "usable with errors". Validation is not necessary for a newly reconstructed block but it provides a safety check. Subsequent normal processing should continually check a block using the same techniques of validation.

Storing the Changed Data

The block must be transferred from central memory to long term storage after the error correction process is completed. The action primarily depends on the current status of the block. If the block is marked "error-free" or "unusable" then it is stored without updating the redundancy. If the block is marked "usable with errors", the block and its redundancy must be updated. Some parts of

the block may have been corrected (if more than one erroneous packet existed in the block).

INTERACTION WITH THE STAFF PERSONNEL

After error analysis is completed on a block, a report should be presented on the staff personnel terminal. The following parameters should be displayed:

- type of block--data base name to which this block belongs
- address of the block on the disk
- status indicator now associated with this block
- data base descriptors where applicable or needed (examples are number of indexed data elements (IDE) in this block along with Data Reference Number (DRN) and occurrence number for each IDE in this block)
- contents of the block--optional since the staff can use the Data String Examine Utility described below. It might be best to go ahead and print the contents of a block marked with a "unusable" or "usable with errors" status indicator since the staff is expected to enter corrections.

Two utilities are needed by the staff to correct errors. The Data String Examine Utility prints the contents of a data string as requested. The Data String Update

Utility allows creation, deletion, or updating of the data string as specified. These utilities have already been implemented in ALS [13].

The following are considerations to incorporate error detection and correction logic.

- These utilities should be available to the staff at any time.
- These utilities must be able to work with blocks marked with any kind of status indicator (including unusable).
- The status indicator should be included in the print of the examine utility.
- Of major importance is an automatic action which would occur as the update is being processed. The status indicator should be set to "correct". New redundancy information should be computed and stored.

SUGGESTIONS FOR BACKGROUND VALIDATION

Validation of many blocks can be achieved by a looping process. Each block would be read from disk. This would automatically invoke error correction if an error is detected. If no error is detected, the block should still be examined for integrity by using the techniques presented in VALIDATION OF ONE DATA BLOCK.

It is suggested that background validators be run regularly in periods of low volume (with a low priority).

This is especially beneficial preceding the generation of a report. Thus the associated data bases will be readied for use. If error correction logic must be invoked, its time and storage requirements will not delay the report. If the staff must be notified of problems that reconstruction could not correct, the staff can have time to develop and enter changes without delaying the report.

If compromises have been made in reduced error detection or correction to increase system performance, the background validator should operate with maximum capabilities. This is especially true if detection ever uses less redundancy than that saved for error correction. For example, detection may only use one checksum for a block, thus failing to detect a class of self-compensating errors. An expanded detection scheme can use parity strings in an algorithm closely paralleling that used for error correction. This use of increased redundancy enables the algorithm to detect more errors.

SUMMARY USING A MODEL OF SYSTEM LOGIC INCLUDING DETECTION AND CORRECTION

A simplified overview of a typical system is needed to show how error detection and correction control the logical flow of data through the system. The model in Figures 2 and 3 is offered as a pictorial aid. This flow is complicated by the degree of error which possibly can occur in the data and the length of time that an error

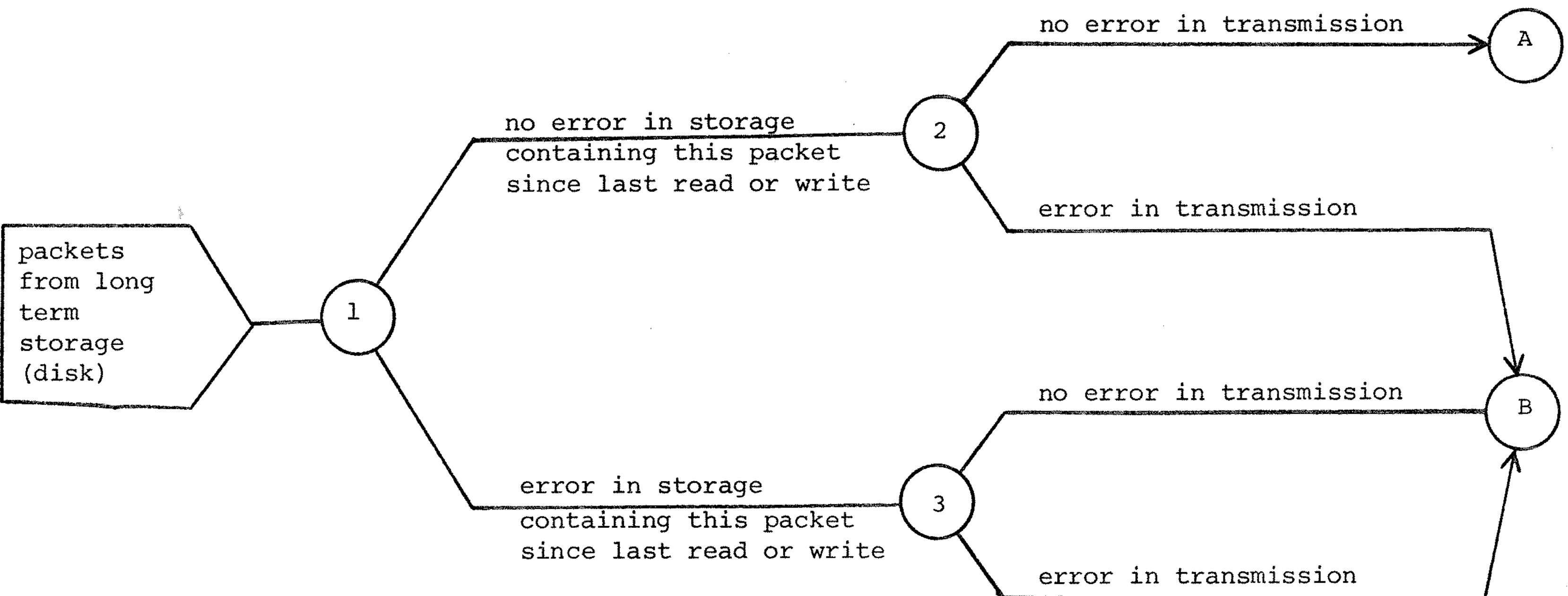

Figure 2: Data Integrity Management Network Model, Introduction of Errors

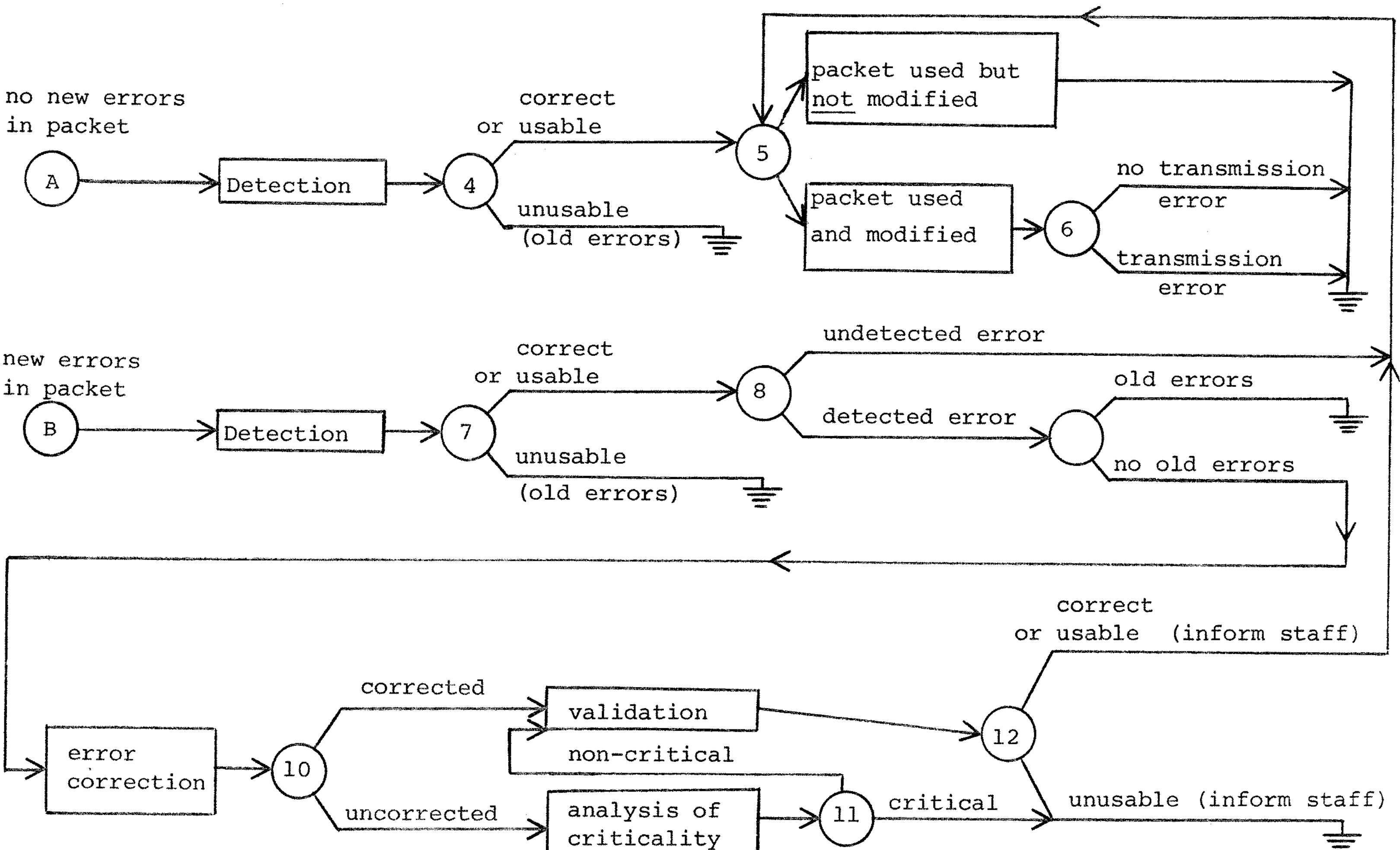

Figure 3: Data Integrity Management Network Model, Processing Errors

may remain in the data.

A packet is a unit of data. A packet comprises one rectangular array of bits of the size chosen to enter the error correction routine at any one time. The possibility of looping through the detector and reconstructor to process a block is not shown in this model.

The model identifies branches within the network by a circled numbered node. At a branch, a packet will enter one of a group of states. Each branching point is associated with a set of probabilities whose sum is equal to one. Each state has a probability p associated with it in the following manner: given that the packet arrives at that branching point, the packet will enter that state with a probability of p.

Figure 2 shows how packets come from long term storage such as disk into central memory. Just considering the time interval since this packet was last read into or written from central memory (i.e. the time interval since the packet was last created, updated, or readied for use) the packet may or may not have experienced errors. Errors are introduced into the data by a storage failure with a probability represented at node 1. Errors are introduced into the data by an Input/Output transmission problem with a probability represented at either node 2 or node 3.

Continuing Figure 2, Figure 3 shows the system flow through the detection unit. Now the main complication

to the logic appears. The degree of error determines whether the data can or cannot be automatically corrected (or re-corrected). The criticality of the data determines whether or not it can be used while uncorrected.

Each packet is marked with a usability indicator regarding old errors (see section on Status Indicator for a Block in DETECTION). A packet is marked usable if it contains a non-critical uncorrected error. A packet is marked unusable if a critical error exists in the data, automatic error correction has failed, and the staff has not yet manually entered corrections.

If a packet from node A goes through the detection unit and, with a probability represented at node 4, is found to be marked with a status indicator of unusable due to old errors, the packet immediately leaves the system. Otherwise the packet is correct or is usable with regard to old errors (and since it is free of new errors), it is used in the normal computations of the system. During normal processing, the packet is used with or without being modified and rewritten to the long term storage. The probability of modification is represented at node 5. Here the Input/Output transmission could introduce errors into the packet with a probability represented at node 6.

If a packet from node B goes through the detection unit and, with a probability represented at node 7, is found to contain a status indicator of unusable, the

packet immediately leaves the system. Otherwise, the packet contains a status indicator of correct or usable and may be tested for new errors (errors which have occurred since the packet was last read or written to long term storage). The new error may be detected or undetected with a probability represented at node 8. If the packet is found to contain new errors, its status indicator must be rechecked. The branching probability is shown at node 9. If the packet has old errors indicated by the status indicator, it is deemed to be experiencing too many problems to be safely used, and it is forced to leave the system. Otherwise the packet has new errors but no old errors and it is sent to the error correction unit.

The packet leaves the error correction routine either as corrected or uncorrected with a probability represented at node 10. A corrected packet goes through validation with a possibility of being found unusable represented at node 12. An unusable packet leaves the system and the staff is informed that further action is necessary. A correct or usable packet joins the path of normal processing at node 5. The staff should be informed. Subsequent manual corrections are expected for a usable packet to make it error-free.

An uncorrected packet leaving node 10 is analyzed for criticality of data content. It is found to be either critical or non-critical with a probability shown at node

11. If it is deemed critical, then it joins the unusable path. A packet containing non-critical data joins the corrected path from node 10. Validation will check for structural validity before this packet will be considered usable.

To return to another possible branch leaving the detection unit at node 8, a packet may contain errors which are undetected. Thus the packet joins normal processing. This is the most critical, dangerous path in the model. The potential cost may be great. If the error affects the structure of the data, the cost involves possible propagation of the error. Degradation of the data and/or the system will incur any degree of cost necessary to repair the system when a malfunction becomes apparent. This can range from reprocessing one interactive request to rollback and restart. On the other hand, if the error involves critical information, the cost is largely intangible--a function of the criticality and the propagation of the error in further processing.

CHAPTER IV

ALGORITHMS FOR ERROR DETECTION AND CORRECTION

PARITY SCHEMES

Parity is an established computer concept in hardware design to detect errors. Generally one additional bit called the parity describes the state of a group of bits. The parity may be either even or odd. In a scheme called even parity the total number of bits turned on counting the parity bit is even. Specifically, if an odd number of bits is on then the parity bit is turned on. If an even number of bits is turned on then the parity is turned off. For example, for the string 0101 containing an even number of bits turned on, the parity would be equal to zero.

Whenever the group is accessed, the parity is recomputed by the hardware and compared with the original stored value. If there is a difference, the group has developed errors. However, the erroneous bits cannot be identified individually. The original cannot be deduced from the present condition.

If such a parity scheme is not included in the hardware design, it can be implemented by software at some increase in storage and processing requirements. More