Applied Research Laboratory

Technical Report

ANALYSIS AND SYNTHESIS OF
ROBUST DATA STRUCTURES

by

A. Ravichandran
K. Kant

PENNSTATE

The Pennsylvania State University
APPLIED RESEARCH LABORATORY
P.O. Box 30
State College, PA 16804

ANALYSIS AND SYNTHESIS OF ROBUST DATA STRUCTURES

by

A. Ravichandran
K. Kant

Technical Report No. TR 90-010
August 1990

Supported by:
Space and Naval Warfare Systems Command

L.R. Hettche, Director
Applied Research Laboratory

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	August 1990	

4. TITLE AND SUBTITLE

ANALYSIS AND SYNTHESIS OF ROBUST DATA STRUCTURES

5. FUNDING NUMBERS

6. AUTHOR(S)

A. Ravichandran, K. Kant

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Applied Research Laboratory
Penn State University
P. O. Box 30
State College, PA 16804

8. PERFORMING ORGANIZATION REPORT NUMBER

TR 90-010

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Space and Naval Warfare Systems Command
Department of the Navy
Washington, DC 20363-5100

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

N-00039-88-C-0051

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

There is an increasing interest in applications in which the reliability of a computing system is of utmost importance. Also, it is likely that the availability of reliable computing systems would promote their use in critical application areas. One approach to increasing the reliability of computer software is by increasing the robustness of data structures used.

In this thesis we provide a formal approach for the analysis and synthesis of robust data structures. The entire data structure is viewed as a collection of data elements related via some attributes. The relationships are specified by a set of axioms in first order logic. Faults in attributes invalidate some of the axioms. The invalidated axioms are used to detect and correct the faulty attributes. We derive sufficient and in many cases necessary conditions for achieving a given level of detectability and correctability. We discuss the notion of compensations and extend our design to tolerate compensating faults.

Keywords: Statistical measurement

14. SUBJECT TERMS

robust data structures, computer reliability, analysis, synthesis detectability of errors, correctability of errors (KR)

15. NUMBER OF PAGES

120

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR

13. Abstract (continued)

We also show how detection and correction can be localized to small portions of the data structure, thereby allowing concurrent repair in several disjoint portions. This property makes local correction attractive for B-trees and other structures used in data bases. We then show how the ideas developed for attaining structural integrity can be applied to achieve data integrity as well.

We then provide an optimal algorithm for the identification of faulty attributes in a robust data structure designed using the method described in the thesis. The algorithm does not use any fault syndrome table since the size of such a table could be large, particularly when faults can compensate one another arbitrarily. We show that the identification is possible in time proportional to the number of axioms even when faults compensate one another arbitrarily. This is optimal, since our method of axiom generation does not yield any redundant axioms.

Abstract

There is an increasing interest in applications in which the reliability of a computing system is of utmost importance. Also, it is likely that the availability of reliable computing systems would promote their use in critical application areas. One approach to increasing the reliability of computer software is by increasing the robustness of data structures used.

In this thesis we provide a formal approach for the analysis and synthesis of robust data structures. The entire data structure is viewed as a collection of data elements related via some *attributes*. The relationships are specified by a set of axioms in first order logic. Faults in attributes invalidate some of the axioms. The invalidated axioms are used to detect and correct the faulty attributes. We derive sufficient and in many cases necessary conditions for achieving a given level of detectability and correctability. We discuss the notion of *compensations* and extend our design to tolerate compensating faults.

We also show how detection and correction can be localized to small portions of the data structure, thereby allowing concurrent repair in several disjoint portions. This property makes local correction attractive for B-trees and other structures used in data bases. We then show how the ideas developed for attaining structural integrity can be applied to achieve data integrity as well. The design of a robust file system that uses the theory described in the thesis is also discussed.

We then provide an optimal algorithm for the identification of faulty attributes in a robust data structure designed using the method described in the thesis. The algorithm does not use any fault syndrome table since the size of such a table could be large, particularly when faults can compensate one another arbitrarily. We show that the identification is possible in time proportional to the number of axioms even when faults compensate one another arbitrarily. This is optimal, since our method of axiom generation does not yield any redundant axioms.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Reliability of a system is a statistical measure that expresses the probability that the system will conform to its specifications [AL81]. We consider a system to be reliable if it is highly probable that it will provide service to our satisfaction. The reliability of a computing system depends on the reliability of both the hardware and software. Hardware reliability has been studied extensively [SS82] and will not be discussed here. We shall confine our attention solely to software reliability.

Avizienis [Avi78] defines two complementary approaches to achieving software reliability: *fault intolerance* and *fault tolerance*. Fault intolerance includes techniques applied during system development to ensure that the running system satisfies all reliability criteria a priori. Examples are correctness proofs, programming methodology, etc. As stated by Parnas [Par77], this approach is based on the following two assumptions:

1. the machine which interprets the software will function properly, and

2. all data inputs to the system will be correct.

Thus this approach cannot cope with residual design flaws, hardware faults or user errors.

The fault tolerant approach attempts to increase the reliability by designing the system to work even if the assumptions stated above do not hold (although the system may provide a lesser level of service). Since design faults cannot be completely eliminated nor can hardware error be ruled out, the fault tolerant approach is better suited for achieving software reliability. Before discussing the fault tolerant approach we define a few concepts which shall form the basis for all further discussions.

1.1 Terminology

Any algorithmic or hardware malfunction is said to be a *fault.* State transitions of the system performed by a faulty component lead to an incorrect state. Such an incorrect state is known as an *erroneous state.* Continuation of the computation from an erroneous state leads to a state which does not satisfy the specifications of the system. In such a case a *failure* is said to have occurred. An *error* is that part of the erroneous state which causes the failure.

From the definitions above, it is clear that a fault need not always lead to a failure. Moreover, the occurrence of a fault is not observed unless it manifests itself as an error. One approach used to obtain software reliability is to avoid failure states altogether. Such a system should possess *error detection* capability, i.e., the ability to discern if the system is in an erroneous state. Furthermore, once the system realizes that it is in an erroneous state, it should take corrective action so that further computation does not result in a failure state. The corrective action taken on detection of the erroneous state is known as *error recovery* or *error correction.* Essentially, error recovery entails state restoration. Error recovery can be classified as *forward error recovery* or *backward error recovery* depending on the technique used for state restoration. Backward error recovery involves restoring a previous error free state of the system as the new state. In contrast forward error recovery techniques manipulate some part of the state information to get a new state. We shall discuss each of these techniques in some detail.

1.2 Forward Error Recovery

This technique involves manipulating the erroneous state in order to get a new consistent error free state. Thus for every erroneous state we need to know the manipulation required to get the new state. Secondly, we should be able to predetermine the set of erroneous states that can be reached during a computation. Cristian and Best [CB81] provide a solution to the second problem. They define the exception occurrence as the event when

computation reaches a state which is inconsistent (erroneous state). They then provide a systematic method for detecting exception occurrences. In order to facilitate this they define exceptions in terms of the weakest preconditions for each operation. Using this method of exception detection Cristian [Cri80] provides a method for state restoration. Each erroneous state is associated with a distinct exception. The exception handling routine then modifies the appropriate state variable to obtain the new state. This method has the advantage of not requiring any additional system resources in order to perform recovery. This method suffers from the drawback of being unable to handle unanticipated faults. Secondly, this method is totally dependent on the system being designed and hence cannot be provided as a general mechanism.

1.3 Backward Error Recovery

Backward error recovery techniques are the result of attempts made to carry many of the hardware techniques to the software arena. Relevant examples in this context are

- multiversion software, which is an adaptation of N-modulo redundancy (NMR) technique.
- recovery blocks, which is an adaptation of standby redundancy technique.
- robust data structure, which is similar in spirit to error correction codes.

These techniques involve restoring a correct state as the new state. All that is necessary for these techniques to work is the availability of a correct state. Restoration of the correct state eliminates all the effects of the fault. These techniques are applicable to every system since the notion of state is present in every system. The most attractive feature of these techniques are that it makes no assumptions about the faults or the environment in which the system operates. In fact this can be used as a general mechanism for any software system.

These techniques, however, are not without drawbacks. The major problem with the backward error recovery approach is the additional storage required for state restoration

and the cost associated with the restoration process itself. The other disadvantage concerns those objects which cannot be restored. This is especially troublesome in a set of communicating processes which may take action based on the inputs received from some other process in the set. However, some techniques for implementing backward error recovery mechanism incorporate features to overcome the abovementioned problems. Backward error recovery has been studied extensively.

1.3.1 Recovery Blocks

Assume that a non-redundant software module has been prepared to perform a task reliably. This module is called the *primary module.* The output of this module is checked to see if the specifications are satisfied. These checks are called *acceptance tests.* If the acceptance test is successful then it is assured that the system has executed correctly. In case of failure of the acceptance test, a second module called the *alternate*, which is designed to perform the same task, is executed. Before the execution of the alternate is started the system state is restored to the prior state in which the primary module started execution. The output of the alternate is then subjected to acceptance test and the above procedure is repeated with the third alternate if necessary. If all the alternates fail the acceptance test then a failure is reported and corrective action needs to be taken. Notice that this scheme caters to any failure as long as the fault does not influence the results of the acceptance test.

Horning [Hor74] was the first to describe the recovery block mechanism. He defined the semantics of recovery blocks and presented an implementation of the recovery cache. Randell [Ran77] presented a recovery block scheme in which he discusses solutions to the problems of error recovery in the presence of interacting processes. He introduced the notion of *conversations* which helped reduce the amount of recovery overheads. Kant and Silberschatz [KS85] highlight some of the basic issues in performing backward error recovery in concurrent systems. They present a set of necessary and sufficient conditions for ensuring that the amount of computation lost by error recovery is bounded. Shrivastava [Shr78]

provided an implementation of recovery blocks in sequential Pascal and later on [Shr79] discussed extensions to the language Concurrent Pascal for implementing recovery blocks. Numerous other authors have discussed other implementation details of recovery blocks and provided performance characteristics of their implementation [And85].

Randell and Smith [MSR77] discuss a mechanism which uses concepts from both forward and backward error recovery technique. They argue in favour of using exception handling mechanism for anticipated fault situations and the recovery block scheme for unanticipated situations. Cristian [Cri80] also arrives at the same conclusion. Kant [Kan87] discusses a scheme that combines forward and backward error recovery techniques for achieving high software reliability in real-time environments. The scheme allows a fairly flexible inter-process communication mechanism. Several authors have suggested language extensions to different languages and provided implementations using these features for such a hybrid approach.

1.3.2 Multiversion Software

Avizienis [AC77] was the first to adapt NMR technique into software. He called this approach N-version programming. In this scheme, N ($N > 1$) versions of a program designed independently to satisfy a common specification are executed and their results compared. Based on majority voting the erroneous results can be eliminated and the correct results passed on. The effectiveness of N-version programming approach depends on the extent of diversity and independence among the different versions. Some studies for exploring the effectiveness of this approach are reported in [Sco84, And85, CLE88]. The issue of independence between the different versions is examined in [EL85, KL86]. Various levels of diversity between versions have also been suggested, for example, different algorithms [CA78], specification languages [KA83] and even programming languages. Other issues such as efficient implementation, hardware support, recovery in concurrent processing environments have also received wide attention.

Both recovery block and N-version programming approach are similar in that they employ redundancy to tolerate software errors. Recovery block approach employs redundancy over time whereas N-version programming approach employs redundancy over space. A comparative evaluation of the N-version programming and recovery blocks is presented by Grnanov et al. [GAA80].

1.3.3 Robust Data Structure

Any software system executes by massaging the data. The data is maintained in some logical organization called the data structure. The approaches discussed until now fail to operate reliably when the data structure on which they operate is corrupted due to the occurrence of a fault. Thus any good fault tolerant system requires the use of a reliable data structure, also referred to as a *robust data structure*. Since data structures are also stored in the same hardware as programs and operated on by software they are also subject to faults. Hence we require some mechanism to ensure the reliability of data structures.

Both N-version programming and recovery block technique depend on *design diversity* for effectiveness. The extent of diversity attainable depends on the dissimilarity of data structures used by the different versions (or alternates). Usage of same or similar data structures in the different versions (or alternates) gives very limited diversity. It would be advantageous to structure the data so that different versions view the logical organization differently. This entails use of redundant information in the data structure. This redundant information can be used to increase the robustness of the data structure in addition to providing independence amongst the different versions.

This thesis will deal with the design of robust data structures. We shall provide a framework in which to study this design and then provide a method to synthesize a robust data structure. Before we proceed to discuss the details, let us familiarize ourselves with other work in the area of robust data structures.

1.4 Literature Survey

Data structure reliability has not been the focus of much research. The first reported work in the area was as late as in 1980. As in the case of fault tolerant software, redundancy is the key to detection, diagnosis and recovery from faults which affect the integrity of the data structure. Taylor et al. [TMB80] were the first to study where and how to apply redundancy in order to obtain fault tolerant data structures. In view of the importance of this work we shall discuss it in some detail.

They define a *data structure* as the logical organization of data. The *storage structure* is a representation of the data structure. A *data structure instance* is defined as a particular instance of a data structure. A *change* is defined as an elementary modification of the data structure instance. They confine their attention to faults that result in changes to structural information of the data structure, i.e. they confine their attention to *structural integrity* and not *semantic integrity.* Structural integrity is concerned with the correctness of the representation of the data. Semantic integrity deals with the meaning of the data.

Using this background they define N-detectability and N-correctability. N-detectability refers to the situation when N or fewer changes in the data structure can be detected. N-correctability refers to the situation when we can recreate the correct instance of data structure modified by N or fewer changes.

They then define three properties of the storage structure representing the data structure. The properties are

ch-same the minimum number of changes that transform a correct instance of the data structure into another correct instance and contains the same set of nodes.

ch-repl the minimum number of changes required to replace one or more nodes in the data structure instance with the same number of nodes from outside the instance so that the number of nodes remains the same.

ch-diff the minimum number of changes required to transform one correct instance to

another correct instance involving a different number of nodes.

Their methodology is quite simple. For any data structure they evaluate *ch-same, ch-repl* and *ch-diff*. The detectability is one less than the minimum of the three. They then add some additional structural entities so as to increase the minimum value. The procedure is repeated till they achieve the desired level of detectability. Based on this scheme they provide the values for ch-same, ch-diff and ch-repl for certain types of data structures. They also provide a relation between the requirements of detectability and correctability.

We state their main result in the form of a lemma.

Lemma 1.4.1 *If a storage structure employing identifier fields is 2r-detectable and there are at least $r+1$ edge disjoint paths to each node of the structure, then the storage structure is r-correctable.*

The drawback of the scheme is that they do not provide a method of adding redundant information systematically. Secondly their model of the data structure and the manner in which faults are considered are too restrictive. They do not provide a good error detection and error correction procedure.

Seth and Muralidhar [SM85] use the same scheme as in Taylor [TMB80]. However, they provide a systematic method for generating the error correction and error detection algorithms.

After Taylor's work, many authors have provided mechanisms to improve the reliability of some particular data structure. Sampiao and Sauve [SS85] provide a method to make the tree data structure more robust. Davis [Dav87] provides a method for correcting an AVL tree. His method, called *local correction*, does correction by checking a small subset of the data structure in the vicinity of the faulty element, hence the name local checking. Recently Taylor and Black [TB86] have provided an implementation of a locally correctable B-tree. The drawback of each of these methods is that they are applicable only to the structure defined by them. It is not at all clear how to extend this method to other structures.

Of late, there has been a growing interest in this area due to the requirement of efficient

concurrent operations in data base systems. There have been a number of papers that deal with the efficient access of data stored in the database. These methods use some additional structural entity on the basic data structure in order to speed up the operations. One such mechanism is described by Lehman et al. [LY81]. They provide a method for efficient concurrent operations on B-trees. Similarly Sagiv [Sag85] provides a method for performing concurrent operations on B-trees where the operations may overtake one another. These works are related to ours since they also deal with providing accessibility to data while some portion of the structure is inconsistent. Here too the authors do not provide a general method for adding structural redundancy.

We propose a general method that enables us to add structural redundancy. The structural redundancy will be used for fault tolerant purposes. The rest of the thesis is organized as follows: A model of the data structure is presented in chapter 2. General results that characterize the amount of redundancy required to achieve the desired level of fault tolerance is presented in chapter 3. A general algorithm for identifying faulty attributes in a data structure designed using our method is the topic of discussion in chapter 4. Some applications of the theory described in this thesis are discussed in chapter 5. Conclusions and directions for future research are discussed in chapter 6.

Chapter 2

Model of Data Structure

We define a data structure D as the triple (EL, AT, AX) where

$EL = \{e_1, e_2, ..., e_M\}$ set of elements.
$AT = \{a_1, a_2, ..., a_N\}$ set of attributes.
$AX = \{S_1, S_2, ..., S_L\}$ set of axioms.

As an example, let us consider the definition of the data structure DLL. DLL is defined as $\{EL_{DLL}, AT_{DLL}, AX_{DLL}\}$, where $AT_{DLL} = \{fp, bp, count, tag\}$. Attribute fp is used as a forward pointer, attribute bp is used as a backward pointer, attribute *count* denotes the number of elements in DLL and attribute *tag* is used as an identifier field. The axiom set AX_{DLL} is as follows:

$$AX_1 : \forall E \in EL_{DLL}[bp(fp(E)) = E \wedge fp(bp(E)) = E] \tag{2.1}$$

$$AX_2 : \forall E \in EL_{DLL}\{[fp^{count}(E) = E] \wedge [tag(E) < tag(fp(E))]\} \tag{2.2}$$

$$AX_3 : \forall E \in EL_{DLL}\{[bp^{count}(E) = E] \wedge [tag(E) > tag(bp(E))]\} \tag{2.3}$$

Figure 2.1 shows a specific instance of data structure DLL, with three elements. In axiom AX_2, $fp^{count}(E) = E$ is a short form of $[fp^n(E) = E \wedge n = count]$, where $fp^n(E)$

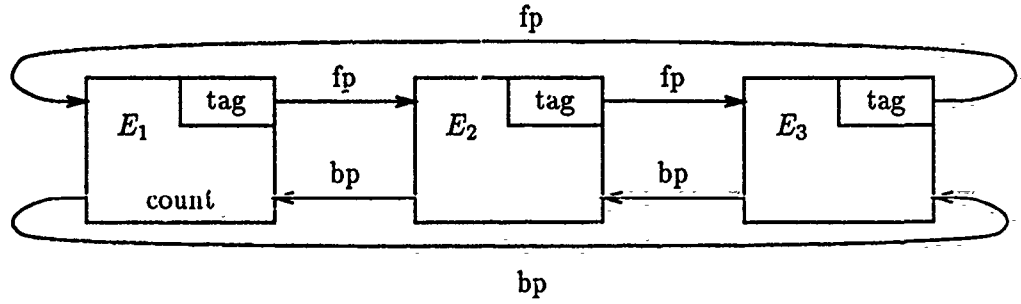

Figure 2.1: An example of a data structure

is $\underbrace{fp(fp(\ldots(fp}_{n\ times}(E)))))$. Axiom AX_3 is obtained in a similar fashion. In any data structure D, the elements represent the collection of data items that D is supposed to store. One of the elements in the set EL of any data structure D is considered to be a distinguished element. The distinguished element is normally referred to as the **header** and we assume that the header is always accessible. In figure 2.1, we assume element E_1 to be the header.

2.1 Attributes

Attributes are entities that provide structure to the collection of elements. Attributes are used to organize the collection of elements and to provide information about them. An attribute could be either **atomic** or **generic**. We call an attribute **atomic** if it pertains to a single element of D or D as a whole. We call it **generic** if it has an instance for every element of D and given an instance of the attribute in D we can access all the other instances of the same attribute in D. The definition of generic attribute should not be construed as the ability to functionally determine the values of all the attribute instances given the value of one instance of the attribute. We require the ability to access at least one other attribute instance from a given attribute instance, which in turn can be used to access yet another instance of the same attribute and so on till all the attribute instances are obtained. In the example data structure DLL, the count of the number of elements is an atomic attribute whereas forward pointer is a generic attribute. A specific instance of a generic attribute (e.g., forward pointer for a specific element) can be regarded as an atomic attribute. An atomic attribute can be thought of as a function that takes a subset of EL as argument and returns another element or a value. In some cases we require a weaker form of the generic attribute which we call as *weak generic* attribute. A *weak generic* attribute has an instance of the attribute in every element of the data structure, but given an instance of such an attribute it is not possible to access another instance of the same attribute in the data structure. The attribute *tag* in the example data structure DLL is an example of a weak generic attribute.

We refer to the representation of the data structure in memory as a *storage structure.* Every element of the data structure is mapped into a set of memory locations in the storage structure. Similarly every attribute is mapped into one or more memory locations in the storage structure. An atomic attribute is mapped to a single memory location whereas a generic attribute is mapped to as many locations as the number of elements.

At this point, it is appropriate to say a few words about the implementation details. The instances of all generic attributes for an element E_i are typically stored along with the data item itself. The atomic attributes that apply to D as a whole are typically stored in the **header.**

At all times the storage structure instance should be a correct representation of the data structure. However, it is not possible to determine the correctness of the representation just by observing the attribute values in the storage structure. We need to exploit the relationships (if any) among the different attributes. These relationships are expressed in the form of axioms.

Axioms are well-formed formulae in first order logic involving functions and predicates over attributes. They implicitly define the set of all correct instances of the data structure. We need to *evaluate* the axioms to determine the correctness of the representation of a storage structure instance. *Evaluation* of an axiom consists of converting the axiom into a logical formula and obtaining the truth value of the resultant logical formula. Conversion to a logical formula is accomplished by replacing each attribute by its value in the storage structure. Each function (predicate) over an attribute is replaced by a function (predicate) over the value of this attribute in the storage structure. If the resultant logical formula is true we say that the axiom evaluates to true or the axiom is *valid.* If the logical formula is false we say that the axiom evaluates to false or the axiom is *invalid* or the axiom is *violated.*

A storage structure instance is a correct representation of the data structure if every axiom is true, otherwise the data structure is said to be *faulty.* Consider a storage structure instance SS_{DLL} of data structure DLL, where elements E_1, E_2 and E_3 are stored at memory locations 100, 200 and 300, respectively. The values of attribute fp in elements $E_1, \ldots, E_3$

are 300, 300 and 100, respectively. Similarly the values of attribute *bp* in elements $E_1, \ldots, E_3$ are 300, 100 and 200, respectively. Attribute *count* has a value of 3 and the values of attribute *tag* in elements $E_1, \ldots, E_3$ are 1, 2 and 3, respectively. Axioms AX_1 and AX_2 are false for storage structure instance SS_{DLL} and hence we say data structure DLL is faulty. A faulty data structure consists of one or more attributes, whose values in the storage structure instance invalidate the relationships expressed by the axioms containing these attributes. Each such attribute is said to be *faulty*. As an example, the attribute *fp* of data structure DLL is faulty in the storage structure instance SS_{DLL}. The attribute values in the storage structure could be incorrect due to a hardware fault or a software error. Also an incorrect value of an attribute in the storage structure could be interpreted in different ways (i.e. each incorrect value could be considered as a distinct fault). However, we ignore these aspects and consider every incorrect value of an attribute in the storage structure instance as the same fault. So, with every attribute a_i, we associate a unique **fault** f_i. Thus, we have the **base fault set** $BFS = \{f_1, f_2, ..., f_N\}$. It is important to note here that the altered value of an attribute in the storage structure is arbitrary and may not even belong to the prescribed domain; therefore, the interpretation of faulty values needs some care. This issue will be revisited later. For generic attributes, we could have two different fault models: (a) **Generic Fault Model (GFM)**, where the fault in an attribute may alter values of an arbitrary number of instances of the attribute, and (b) **Instance Fault Model (IFM)**, where only one instance is affected. Consider a storage structure instance of *DLL* with two erroneous attribute values, one in *fp* of element E_1 and the other in *fp* of element E_2. Under **GFM** the two erroneous values are viewed as a single fault in *fp*, whereas under **IFM** the same two erroneous values are considered as two distinct faults. We shall mostly deal with GFM in this thesis, because IFM becomes a trivial special case of GFM.

Let *FS* denote the set of all possible simultaneous faults. Then,

$$FS = powerset(BFS) - \emptyset = \bigcup_{i=1}^{N} F_i \tag{2.4}$$

where F_i denotes the set of all possible i simultaneous faults. We use $f_1, f_2, \ldots$ to refer to faults in the base fault set and $f_x, f_y, \ldots$ to refer to multiple faults. In case of a specific multiple fault, we replace the subscript x, y by a string of attribute names. We use f, g to refer to faults which belong to either of the two categories.

2.2 Axiom Structure

The effect of a fault on the attributes of D is captured by the violation of the axioms that involve one or more of the faulty attributes. In other words, the identification of a faulty attribute is done on the basis of axioms that are violated. To do the identification of faulty attributes, the axioms must satisfy several restrictions. Also, in order to obtain results concerning the number and type of axioms needed for a given level of fault tolerance, we need to give the axioms some structure. To that end, we express all axioms in conjunctive normal form with all quantifiers pushed to the outermost level. We then collect all conjuncts that involve the same set of attributes, and call them a **term**. Thus an axiom will be a conjunction of some terms, each one of which would involve a distinct set of attributes. For example consider three attributes p, q and r and predicates $A(p,q)$, $B(p,q)$ and $C(q,r)$. We express these predicates in an axiom as follows:

$$A(p,q) \wedge B(p,q) \wedge C(q,r) \tag{2.5}$$

Since the first two conjuncts involve the same set of attributes, namely p and q, we combine them into a term and thus obtain the axiom as

$$A'(p,q) \wedge C(q,r) \tag{2.6}$$

where $A'(p,q) = A(p,q) \wedge B(p,q)$. We now introduce the following definitions:

Definition 2.2.1 *Let A be the set of attributes comprising axiom S, F be the set of single faults in the attributes of the set A and let exactly one attribute in the set A be faulty. Then S is said to be* **proper** *if for every instance of D, the axiom S evaluates to false for all $f \in F$. (Notice that we do not say anything if more than one attribute in S is faulty).*

Consider a modified version of data structure DLL with axiom AX_2 replaced by axiom AX_2' where

$$AX_2' : \forall E \in EL_{DLL}[fp^{count}(E) = E] \tag{2.7}$$

In a correct instance of DLL, the values of attribute fp in elements $E_1, \ldots, E_3$ should be the addresses of E_2, E_3 and E_1, respectively. Consider an instance of DLL, where as a result of a fault in fp, the values of attribute fp in elements $E_1, \ldots, E_3$ are the address of E_3, E_2 and E_1, respectively. Axiom AX_2' is true in this case also whereas axiom AX_1 is false. Axiom AX_2' is unable to detect permutation of the values of the attribute fp and hence is not proper. To make AX_2' proper we modify it to obtain AX_2.

Definition 2.2.2 *The axioms are said to be in* **Attribute Normal Form (ANF)** *if every term of every axiom involves the same number of attributes and a term does not appear in more than one axiom.*

Definition 2.2.3 *An axiom S is said to be in* **(q,k)-ANF** *if it is in ANF, each term of S has exactly q attributes and S involves at most k attributes. If $q = k$, we call the form simply* **k-ANF**.

We now state the restrictions on the form of axioms.

1. Any axiom involving a generic attribute is **universally quantified** over all its instances. Informally, this means that any property of a generic attribute involves every instance in a uniform way (i.e. none of the instances is *special*).

2. Each axiom is proper. Furthermore, no subset of terms of an axiom is proper. This is essential for fault identification based on violated axioms.

3. The set AX is (q,k)-ANF for some given q and k. The underlying motivation for this is to ensure that the number of axioms and the number of attributes used in an axiom become significant. The requirement of exactly q attributes per term is not restrictive, since we can always add predicates to the terms to have the same number of attributes

in each of them. In this thesis we only consider terms having two attributes, i.e. we consider axioms in (2,k)-ANF.

4. The axioms are sound and complete. Soundness means that any property derived from them about D is true. Completeness means that they collectively admit only the correct instances of D; that is, any true property of D is derivable from them.[1]

It is important to note at this point that we do not require the axioms to be independent; in fact, the redundancy introduced by the lack of independence is essential for the identification of faulty attributes.

We associate with every axiom S_i a set A_i which is the set of attributes contained in that axiom. We denote the number of attributes in S_i as $|A_i|$.

2.3 Detectability and Correctability

We now define the **failure function** of a fault f, denoted $h(f)$, as the set of axioms that remain valid under f. Then we have the following sufficient (but not necessary) conditions for detectability and correctability:

- A data structure D is **m-detectable** if the failure function of any i-ary, fault, $i \leq m$, is non empty and is different from the set AX, i. e. ,

$$\forall f \in F_i, 1 \leq i \leq m, [h(f) \neq AX, h(f) \neq \emptyset] \tag{2.8}$$

- A data structure D is **m-correctable** if
  - At least m faults are detectable,
  - The failure function for any two distinct faults are distinct, i. e. , $\forall \psi_1 \in F_i, \psi_2 \in F_j, 1 \leq i,j \leq m, h(\psi_1) \neq h(\psi_2)$, and

[1] In practice however, it may sometimes be simpler and more efficient to account for some properties implicitly (and leave the axioms incomplete to that extent).

– The failure function of every fault of arity less than or equal to m contains an axiom involving a generic attribute.

In characterizing m-detectability, we have included the condition $h(f) \neq \emptyset$ to enable distinction between faults of arity $\leq m$ from those of arity $> m$. However, m-detectability does not require the ability to distinguish between two faults of arity less than or equal to m. Also, note that the characterization of m-correctability only ensures that we can uniquely identify the faulty attributes. In principle, this is adequate since we have assumed that any values of the faulty attributes (along with the existing values of other non-faulty attributes) that satisfy all axioms in AX, is a correct representation of the data structure. Unfortunately, no general algorithm is possible for finding such a set of values for $a_1, ..., a_i$; thus the design of correction procedures would require some further information about the specific data structure under consideration. We shall comment on this aspect in the examples.

One issue not addressed above is the **compensation** among faults. Compensation refers to the situation where two (or more) simultaneous *faults* are such that some axiom involving them evaluates as true. These *faults* could well be **internal** to a generic attribute (i.e., in different instances of the same generic attribute) or external to the attributes (i.e. in instances of different attributes). Thus we need to consider both internal and external compensations. We reserve discussion on both types of compensation for section 3.3.3. Thus, for now, we assume that no compensations take place.

When a new attribute is to be added, we need some guidelines to help us in the choice. One general criterion, obviously is to use an attribute that requires minimum storage. Duplication of certain attributes (e.g., count of number of elements, etc.) is sometimes useful. Having more than two copies of the same attribute is however not useful since discrepancies amongst the copies makes it difficult to distinguish the correct copy from an incorrect one. We also need to decide if the new attribute should be atomic or generic. This is easy since $m + 1$ generic attributes are necessary and sufficient for m-correctability.

Our discussion until now places no restrictions on the value of either q, the number of attributes in a term or k, the number of attributes in an axiom, except those resulting from the requirement that the axioms be proper. In this thesis we shall restrict our attention to the situation when each term is comprised of two attributes, i.e. $q = 2$. Such a restriction helps simplify understanding of our approach. Moreover, $q = 2$ is sufficient to express properties relating attributes for most of the data structures of interest. It is advantageous to choose q as small as possible for several reasons:

1. Fewer attributes per term give better detectability and correctability since the term (and hence the axiom containing it) will be violated in fewer cases.

2. With fewer attributes per term, there are fewer possibilities for external compensation.

Some of the terms in an axiom may initially consist of only one attribute. To satisfy the $(2, k)$-ANF requirement, it would appear that we have to artificially convert these to two attribute terms. However, terms with one attribute only specify range restrictions, which must be enforced anyhow for proper interpretation of faulty values, as discussed in the next section. Thus for our formal analysis, we can simply ignore such terms. We shall also fix k in our analysis, and let N (the total number of attributes) be determined by the values of q and k.

In view of the above, we shall impose the following discipline in our design of robust data structures. We start by generating terms for an axiom. We keep adding terms to an axiom until the axiom becomes proper. If necessary, we generate additional axioms using the existing attributes without violating the ANF criterion (i.e. no term appears in more than one axiom). If we have exhausted all possibilities of generating either more axioms or more terms for an axiom with the existing set of attributes then we add an additional attribute.

2.4 Interpretation of Values for Faulty Attributes

Until now, we have tacitly assumed that the value of a faulty attribute always lies within the prescribed domain. Unfortunately, this is not true in general, and may lead to both theoretical and implementation problems. For example, in a circular linked-list, the domain for the pointer attribute is the set of starting addresses of all the elements. If, however, the pointer value changes to some other value, we face two difficulties with respect to the axioms involving this pointer: (a) the axioms may become undefined, and (b) if the location being pointed to is outside the accessible address space, the evaluation of the axioms will cause exceptions. Similar problems could arise with non-pointer attributes; for example, the count of the number of elements in the list could become negative and thereby make certain operations (e.g., mod function, indexing into an array, etc.) undefined and exception causing. In the following, we address these issues briefly.

One way to handle the theoretical problems is to extend the framework with the undefined element $\perp$. One complication introduced by such an extension is that faults that result in undefined values must be distinguished from those that don't. We avoid the added complexity by simply **mapping** all undefined values to some *special* value(s) within the domain. For example, any undefined pointer value could be considered to point to itself. A side effect of such a convention is that valid pointers cannot point to themselves. This is usually not difficult to arrange. For example, one could always retain at least one (dummy) element in a circular-list so that the pointer from the header never has to point to itself. Similarly, all negative values for the count attribute could be mapped to value 0.

This mapping must be supported by the implementation. That is, given a value, we should be able to check if it lies in the desired domain, and if not, map it to the distinguished value. This could either be done explicitly or by an exception handling mechanism. In either case, if the domain consists of a set of noncontiguous values, the checking may be expensive. For example, the domain of links in a linked data structure must explicitly specify the starting addresses of all the allocated elements. Unfortunately, one cannot maintain such

information without using a linked data structure. A reasonable alternative, as suggested in [TMB80], is to assume that every element can be tagged in some way by the identity of the data structure (and user) it belongs to. However, if the pointers are simulated by array indices (as would be done when implementing linked structures in, say, Fortran), domain checking problem becomes much simpler. Finally, note that since m correctability requires at least $m+1$ generic attributes, the nonfaulty attribute can always be used to trace all the elements that belong to the structure.

For purposes of analysis we require a convenient representation of the data structure model. We now present such a representation.

2.5 Hypergraph Model

We represent the data structure D by means of a Term graph (TG_D) and an Axiom hypergraph (AG_D). In both the graphs, the vertices represent the attributes and may be labeled by attribute names. For axioms in $(2,k)$-ANF each pair of attributes (a_i, a_j) present in a term is represented as an edge $a_i a_j$ in the term graph TG_D. An axiom is represented as a hyperedge in the axiom graph AG_D. We can view the edges in TG_D as giving the internal structure of the hyperedges of AG_D. We say that the axiom graph has a connectivity m if every node participates in at least m distinct edges.[2]

Let us discuss the structure of the hyperedges of an Axiom Graph. We impose certain restrictions on the structure of hyperedges. The imposition of a structure on the hyperedges is based on the information content of the three attribute types. Of the three attribute types, a generic attribute has the maximum information content. This is because there is an instance of the attribute in every element of the data structure and secondly, given an instance of the attribute, it provides the ability to obtain another instance of the attribute. An atomic attribute provides lesser information than a generic attribute since it pertains to a single element or lumps the property of the entire data structure. Similarly a weak generic

[2]Note that the connectivity as defined here is different from the usual graph theoretic definition.

attribute provides lesser information than a generic attribute, since given an instance of the attribute it is unable to obtain another instance of the attribute. Clearly, the information content of the weak generic attribute and an atomic attribute are incomparable. Secondly, an axiom consists of one or more terms and each term consists of exactly two attributes. The structure of the hyperedge should represent this information adequately. Finally, it is clear that one cannot add a term relating a weak generic attribute and an atomic attribute. Such a term cannot be evaluated, since there is an instance of the weak generic attribute in every element and it is not possible to traverse through every element of the data structure without a generic attribute.

Thus we impose restrictions on the number and the type of the attributes that can be present in the same hyperedge. These restrictions are best understood by viewing each hyperedge as a hierarchical structure where the hierarchy among the attributes constituting the hyperedge is determined by the information content of the different attribute types, as discussed above. Since the terminology associated with trees is very convenient for discussing hierarchical relationships we use this terminology in the rest of this section. We require every hyperedge to be a tree of height 1, i.e., a tree with one root node and the rest leaf nodes, connected to the root node. Since each term has two attributes, each branch of the tree corresponds to a term.

If the axiom has only one term, then either both the root and the leaf could be atomic or both could be generic. In all other cases the root should be a generic attribute and the leaves could be either atomic or weak generic. Note that by imposing such a structural requirement we consider only a subset of the allowed axioms. Although each term can have only two attributes, terms of the form $g(a_1, a_2, a_3)$ (i.e., terms having three or more attributes) can be used provided one of the attributes is generic and this term can be written in the form of the axiom structure discussed above. This still does not take care of terms with three or more attributes all of which are atomic.

Consider axioms AX_2 and AX_3 of data structure DLL. Each of these axioms consists of three attributes and will be represented as a hyperedge in the hypergraph representing data

structure DLL. Each axiom has one generic attribute, one atomic attribute and one weak generic attribute. We impose the structural restrictions on the hyperedge corresponding to each axiom. We thus require the generic attribute to be the root of the tree and the weak generic and atomic attribute to be the leaves. Thus the generic attribute must be present in both the terms of axioms AX_2 and AX_3.

In the axiom graph a fault in an attribute is represented by the removal of all vertices corresponding to the faulty attributes and all hyperedges incident on them. The resulting graph is called the **reduced axiom graph** or Axiom fault graph. The Axiom fault graph for any fault f, of arity up to m, is denoted as $AG_D(f)$. We also define the **reduced term graph** or Term fault graph, denoted $TG_D(f)$, for each fault f. This is done by simply removing all the faulty nodes and the corresponding edges from TG_D. Because the axioms are proper, at least one term of each axiom containing a faulty attribute must be violated. Thus the conditions for detectability and correctability stated earlier can be expressed as follows:

- D is m-detectable if and only if $AG_D(f)$ is nonnull and is a proper subgraph of AG_D for any i-ary fault f, $i \leq m$.
- D is m-correctable if for every distinct i-ary fault $f_i, i \leq m$, either $AG_D(f_i)$ is a distinct, nonnull proper subgraph of AG_D or for two distinct faults f_i and f_j with with same subgraph of AG_D, $TG_D(f_i)$ and $TG_D(f_j)$ are distinct nonnull subgraphs of TG_D.

It is important to note, however, that unlike $AG_D(f), TG_D(f)$ is not unique for a given fault f. This is because the individual terms of an axiom need not be proper and thus may not always be violated. Thus we have a set of $TG_D(f)$'s corresponding to the fault f. We can characterize detectability and correctability solely with respect to term fault graphs. Such a characterization is of little use since the number of term fault graphs is enormous. We need some alternate characterization which is easy to compute. For the present, we shall ignore term graphs altogether and work with axiom graphs only. We return to term

graphs later on in the section dealing with internal compensation.

If the number of attributes in an axiom is two ($k = 2$), then the hypergraph AG_D reduces to an ordinary graph. Moreover, AG_D is identical to TG_D and we can use either one of them.

We introduce the results by considering (2,2)-ANF axioms. The results extend trivially to the case when the hyperedges have the tree structure discussed earlier. In this case, the additional terms are added to make the axioms proper (as seen in the later chapters). The addition does not change any of the results obtained for the (2,2)-ANF case. Henceforth, whenever we use the word graph, we refer to the axiom graph, which is denoted as G_D.

Chapter 3

Characterization of Detectability and Correctability

In this chapter we study the characterization of detectability and correctability. We also provide a method to obtain a data structure possessing the desired level of detectability/ correctability. Throughout this chapter we assume that the data structure under study is represented by the hypergraph discussed in the previous chapter. We first discuss detectability, where we confine the discussion to the $k = 2$ case.

3.1 Detectability Characterization

The detectability of a data structure represented by the graph can be obtained by a straight-forward application of the definition of detectability. The more interesting aspect of study is the synthesis of a data structure for a given level of detectability. Before we provide an algorithm to synthesize a data structure for a required level of detectability, we make a few observations that characterize detectability of a data structure represented by such a graph.

Observation 3.1.1 *If the graph representing a data structure contains an isolated vertex then the data structure has zero detectability.*

Proof: Assume that the graph has an isolated vertex, say a. Let there be a single fault in the system and further let the fault affect vertex a. Since a is isolated the fault in a does not result in the removal of any edges and the fault graph is the same as the original graph. Thus from the definition of detectability, the system is zero detectable. □

Observation 3.1.2 *The maximum detectability of a data structure represented by a graph* $G = (V, E)$ *is* $|V| - 2$.

Proof: Each vertex has maximum connectivity when the graph is complete. Detectability requires the presence of at least one edge in the fault graph. Since faults in attributes correspond to removal of vertices and the edges incident on the vertices, the maximum number of vertices that can be removed from a complete graph without making the graph empty is $|V| - 2$, from which the result follows. □

3.2 Synthesizing m-detectable Data Structures

The algorithm to synthesize a m detectable data structure is based on the concept introduced in the next lemma.

Lemma 3.2.1 *The detectability (det_G) of a system represented by graph G having l connected components is* $\sum_{i=1}^{l}(det_i + 1) - 1$.

Proof: trivial. □

We now provide an algorithm to synthesize a m-detectable data structure given N attributes. The algorithm works as follows. Given a set of vertices the algorithm partitions the vertices among a set of complete components. By putting the vertices in different components, the effect of a fault in an attribute is confined to the component to which the vertex belongs. Each component is made complete to enhance its detectability, thereby enhancing the detectability of the graph. Also, by keeping each component small, fewer edges are required to make the component complete.

The algorithm starts by constructing components containing two vertices each. It then attempts to build the graph of the required detectability by forming bigger components by reducing the number of components and rearranging the vertices among the reduced set of components. The complete algorithm is described below.

Algorithm 1 *Algorithm to synthesize a m-detectable data structure given N attributes.*

Input:
Number of attributes N
Desired level of detectability m
Output:

```
        A graph representation of m detectable data structure
begin
    Construct a graph G with ⌊N/2⌋ components by having two vertices
    per component and an edge joining the two vertices of a component;
    if N is odd then
      begin
          add the remaining vertex, say x, to a component, say C, containing vertices y and z;
          add the edge xy (or xz) to graph G;
          mark component C incomplete
      end ;
    /* let noofcomp denote the number of components in the graph */
    /* let det_G denote the detectability of graph G */
    noofcomp ← ⌊N/2⌋;
    det_G ← noofcomp −1;
    if det_G ≥ m then return G;
      /* G is a graph capable of detecting ≤ m faults */
    else
    begin
        if there is an incomplete component then
          begin
              add an edge to make component complete;
              det_G ← det_G + 1;
          end
        /* at this juncture all components are complete (or cliques) */
        if det_G ≥ m then return G
          else
             begin
                 while (noofcomp > 1) and (det_G < m) do
                     begin
                         select a component, say C, of smallest size, say l;
                         breakup C into l vertices by removing all edges of C;
                         noofcomp ← noofcomp −1;
                         det_G ← det_G − l + 1;
                         while l >noofcomp do
                             begin
                                 add one vertex to each component;
                                 add edges to make each component complete;
                                 det_G ← det_G+ noofcomp;
                                 l ← l− noofcomp;
                             end ;
                         add one vertex to l smallest components;
                         add edges to make each component complete;
                         det_G ← det_G + l
                     end
                 if det_G < m then write ( not possible to obtain desired detectability )
                   else return G;
```

```
            end
    end
end .
```

3.2.1 Correctness of the Algorithm

We now prove the correctness of the algorithm. The first order of business is to show that the algorithm makes progress and eventually terminates. The following lemma states a result in that direction.

Lemma 3.2.2 *Every iteration of the while loop increases the detectability of the graph by one.*

Proof: In every iteration we break the smallest component and distribute its nodes among other components. Let a component, say C, of size q be destroyed. The detectability of C is $q-2$. The reduction in detectability of the graph as a result of destruction of C is $q-2+1 = q-1$ (the plus one factor arises due to the reduction in number of components). Distribution of the q nodes among other nodes increases the detectability of the graph by q. This is because the addition of a node to a component (and the concomitant edges) enhances the detectability of that component by one. Thus the net change in detectability of G is $q-(q-1) = 1$, which proves the lemma. □

It is easy to see that the algorithm obtains a graph possessing the desired level of detectability, From lemma 3.2.2, it follows that every iteration of while loop increases the detectability of the graph. It can also be seen that every iteration of while loop reduces the number of components by one. Since the number of components is finite to begin with and so is the desired level of detectability, the algorithm terminates in a finite number of steps. If the detectability of the graph is less than the required detectability upon termination of the algorithm, then the required detectability cannot be obtained, given the input constraints. This is because, on termination the graph has one fully connected component. From observation 3.1.2 it follows that the maximum detectability of a graph

with V vertices is $|V|-2$. In all other cases, the resulting graph has the required detectability at termination.

We now show that the graph generated uses the fewest number of edges. We need some additional results which we prove in the next couple of lemmas. In the first lemma we show that any rearrangement of the vertices amongst the components in the graph does not enhance the detectability of the graph.

Lemma 3.2.3 *Any transformation of a graph with l complete components to another graph with the same number of complete components does not result in any increase in the detectability of the graph.*

Proof: Consider a graph G with l complete components. Graph G is transformed to another graph G' by removing vertices from one or more components and adding them to other components. For simplicity of proof we only consider the case when vertices are removed from one component and distributed among others (the proof for the other case is an easy extension of this proof). Let q vertices be removed from component C. Since C was complete, removal of q vertices results in the reduction of component C's detectability by q. Let p vertices, $p \leq q$, be added to a component, say C'. Since C' is complete before and after the addition of p vertices, the detectability of component C' is enhanced by p. In general, the detectability of a component, which is complete before and after the addition of some vertices, is enhanced by an amount equal to the number of additional vertices. Since at most q vertices have been added to the different components the increase in detectability of G is q. Thus in graph G', the detectability of component C is reduced by q and the detectability of a set of components is increased by q. Thus the net change in detectability is zero which proves the lemma. □

Corollary 3.2.1 *Let G, consisting of complete components, be a graph with the fewest edges for a certain level of detectability. The difference in the size of any two components of G is at most one.*

Proof: *By contradiction*: Assume that graph G contains two components, say C_1 and C_2, whose sizes differ by more than one, say two. Let C_1 have p vertices, C_2 have $p-2$ vertices and all the other components have $p-1$ vertices. Transform G into another graph G', consisting of complete components, such that component C_1 has $p-1$ vertices, component C_2 has $p-1$ vertices and all other components remain the same. From lemma 3.2.3 it follows that G' has the same detectability as G. However, G' has fewer edges than G since the transformation results in the removal of one vertex and $p-1$ edges from component C_1 and the addition of one vertex and $p-2$ edges to component C_2. But this contradicts the statement of the corollary stating that G is the graph having the fewest edges for a given level of detectability. Thus the difference in size of any two components cannot exceed one, which proves the result. □

Lemma 3.2.4 *The merging of two complete components enhances the detectability of the resulting graph by one if and only if the merged component is complete.*

Proof: *only if*: Consider a graph G with two complete components C_1 and C_2 of sizes k_1 and k_2 respectively. Let C_1 and C_2 be combined into a new component, say C. As a result of combining C_1 and C_2 the number of components is reduced by one. The total amount of detectability lost is k_1-2+k_2-2+1. Since after combination the detectability of G is increased by one, we require component C to have a detectability of k_1+k_2-2. The size of component C is k_1+k_2. Thus from observation 3.1.2 it follows that component C should be complete, which proves the lemma.

if: Let graph G with complete components C_1 and C_2, of sizes k_1 and k_2 respectively, be transformed to graph G' with complete component C', where component C' is obtained by merging components C_1 and C_2. The other components of G and G' are identical. The size of component C' is k_1+k_2 and detectability of component C' is k_1+k_2-2. Thus the detectability of graph G' is $det_G+(k_1+k_2-2)-(k_1-2)-(k_2-2)-1$, i. e. , $det_{G'} = det_G+1$, which proves the result. □

The lemma given above provides one method for enhancing the detectability. However, our algorithm does not use this method since it results in more then the optimal number of edges necessary for a given level of detectability. This leads us to the result stated in the next lemma.

Lemma 3.2.5 *The algorithm to obtain a graph for a given level of detectability uses the fewest number of edges.*

Proof: Given a system with N vertices and a detectability of less than $\lfloor \frac{N}{2} \rfloor$, it follows that the algorithm generates a graph that uses the fewest number of edges. This follows from observation 3.1.1 and the fact that in the generated graph, each vertex has one incident edge (if N is odd then only one vertex has two edges incident on it). If N is odd and the required detectability is $\lfloor \frac{N}{2} \rfloor$ then the algorithm generates a graph with $\lfloor \frac{N}{2} \rfloor$ components in which all but three vertices have one edge incident on them. All the three vertices with more than one incident edge belong to the same component, say C. The detectability of the graph excluding component C is $\lfloor \frac{N}{2} \rfloor - 2$ (follows from lemma 3.2.1. Since component C should have a detectability of one, it follows from observation 3.1.2 that component C be complete. Since the algorithm generates such as graph, it uses the fewest number of edges in this case also.

We now need to show that the algorithm uses fewest number of edges when the desired detectability is more than $\lfloor \frac{N}{2} \rfloor$. Since the algorithm increases the detectability of a graph, composed of complete components and having the fewest number of edges, by one in every iteration of the while loop we need to show that this increase is achieved by the minimum number of additional edges. As outlined in lemma 3.2.3 no rearrangement of the vertices amongst the components, without a change in the number of components itself, results in a change in the detectability of the graph. As outlined in lemma 3.2.4, the detectability of the graph can be increased by one by merging two complete components. Let the size of the component formed by combining two components be $2V$ and the size of each merged component be V. Since the new component needs to be complete (follows from lemma 3.2.4),

we need to retain all the edges in each of the two components and require some more edges. The additional number of edges required, say e, is

$$e = \frac{2V \times (2V-1)}{2} - \frac{2 \times V(V-1)}{2} = V^2. \tag{3.1}$$

The method used in the algorithm results in breaking the clique of size V and increasing the detectability of other components. We have two cases to consider

case i- the number of components is greater than V: In this case the algorithm distributes the V vertices of the broken clique among V other components, increasing the detectability of each by one. Now the sizes of any two components differ by at most one (follows from corollary 3.2.1). Let there be at least one other clique of size V (this is to maintain consistency, since in the previous paragraph we had assumed that two cliques of size V are combined to form a clique of size $2V$). The number of edges added to a component to increase its detectability by one is equal to the size of the component before the addition of the new vertex. Thus the number of edges added in this case is

$$V + (V-1)(V+1). \tag{3.2}$$

This expression is obtained using the fact that we add V edges to a clique of size V, $V+1$ edges to a clique of size $V+1$ and the fact that the V vertices of the broken clique are distributed among one component of size V and $(V-1)$ components of size $V+1$. The number of edges removed due to breakup of the clique of size V is $\frac{V(V-1)}{2}$. Thus the number of edges required additionally is

$$V + (V-1)(V+1) - \frac{V(V-1)}{2} \tag{3.3}$$

$$= V + V^2 - 1 - \frac{V^2}{2} + \frac{V}{2} = \frac{V^2}{2} + \frac{3V}{2} - 1. \tag{3.4}$$

Since the size of a clique is at least two (i.e. $V \geq 2$) $\frac{V^2}{2} + \frac{3V}{2} - 1 \leq V^2$. Hence the algorithm uses the fewest number of edges in this case.

case ii- the number of components, say l is less than V: In this case the algorithm divides the V vertices among l components. Let us assume that the detectability of each component

is increased by r (i.e. by adding r vertices to each component). Again, let us assume that one component has V vertices and the others $V+1$ vertices. The number of edges added to the component with V vertices to increase its detectability by r is $V+(V+1)+(V+2)+\ldots+(V+r-1) = rV+\frac{r(r-1)}{2}$. Similarly, the number of edges added to a component with $V+1$ vertices to increase its detectability by r is $(V+1)+(V+2)+\ldots+(V+r) = rV+\frac{r(r+1)}{2}$.

The total number of edges added is

$$rV+\frac{r(r-1)}{2}+[rV+\frac{r(r+1)}{2}]\times(l-1) \tag{3.5}$$

$$= rV+\frac{r(r-1)}{2}+(l-1)rV+\frac{(l-1)r(r+1)}{2} \tag{3.6}$$

$$= rV(l-1+1)+\frac{r}{2}[r-1+lr-r+l-1] \tag{3.7}$$

$$= rVl+\frac{r}{2}[lr+l-2] \tag{3.8}$$

$$= V^2+\frac{Vr}{2}+\frac{V}{2}-r\,(usingV=rl) \tag{3.9}$$

The number of edges deleted due to breakup of clique of size V is $\frac{V^2}{2}-\frac{V}{2}$. Thus the number of additional edges is

$$V^2+\frac{Vr}{2}+\frac{V}{2}-r-\frac{V^2}{2}+\frac{V}{2} \tag{3.10}$$

$$= \frac{V^2}{2}+\frac{V}{2}(r+1+1)-r \tag{3.11}$$

$$= \frac{V^2}{2}+\frac{V}{2}(r+2)-r \tag{3.12}$$

It is easy to see that for $1 \leq r \leq V$, this quantity is less than or equal to V^2. Hence the algorithm generates a graph which uses the fewest number of edges for a given level of detectability. □

3.3 Correctability Characterization

We begin by providing the general approach to designing correctable data structures. We start with an initial data structure specification via a sound, proper, and complete set of ANF axioms. We then find the failure functions corresponding to all allowed faults.

Two faults are indistinguishable if their failure functions are identical. The set of all indistinguishable faults form an equivalence class, henceforth called the **disambiguation set.** Correctability requires that all disambiguation sets contain but one member; this is achieved by using additional axioms, if necessary. We may have to add more attributes before these axioms can be generated. We shall see requirements on the number of axioms and the attributes they are composed of.

3.3.1 Design of 1-Correctable Structures

In this section we discuss how to get a 1-correctable structure starting with some initial specification of the data structure. General results that tell a priori the required number of attributes and axioms are deferred until section 3.3.2.

3.3.1.1 Systematic Disambiguation

Suppose that the initial specification has some disambiguation sets with cardinality larger than 1. We start by characterizing these.

Observation 3.3.1 *Let W be a subset of AT such that every axiom either involves all attributes from W or none of them, i.e. $\forall 1 \leq i \leq L, (W \subseteq A_i$ or $W \cap A_i = \emptyset)$. Then all faults in $\omega(W)$ are indistinguishable.*

Observation 3.3.2 *Let W_{max} be a set as in Observation 3.3.1 but with maximum cardinality. Then $|W_{max}| \leq k$ (where k is the parameter in (q,k)-ANF requirement).*

Let $IFS = [f_{i_1}, ..., f_{i_n}]$, $1 \leq i_1, ..., i_n \leq N$ denote some set of indistinguishable faults. Let $W = \omega^{-1}(IFS)$. If $n = N$, i.e., if all faults are indistinguishable, by observation 3.3.2, we must have $N = k$, which means that no disambiguation is possible without adding another attribute. Otherwise, we can add an axiom S_{L+1} formed using $1 \leq n_1 \leq |W| - 1$ attributes from W and at least one from $AT - W$. An important property about axiom S_{L+1} is stated in the form of a lemma.

Lemma 3.3.1 *S_{L+1} reduces the disambiguation set size.*

Proof: Since S_{L+1} contains at least one but not all attributes from W, it will remain valid for some of the faults in $f_{i_1}, \ldots, f_{i_n}$, but not all of them, thus reducing the disambiguation set size. □

Let $X_1, X_2, ..., X_p$ be all the disambiguation sets with cardinality larger than 1. Obviously, all X_i's must be disjoint. Without loss of generality, we can assume that $|X_i| \geq |X_{i+1}|$ for all i. Let A_{L+1} be the set of attributes in axiom S_{L+1}. Let $A_{L+1} = \{Y_0, Y_1, ..., Y_p\}$ where $Y_0 \subset AT - \bigcup_{i=1}^{p} \omega^{-1}(X_i)$ and $Y_i \subset \omega^{-1}(X_i), 1 \leq i \leq p$ Then to achieve maximum reduction in disambiguation set size by adding S_{L+1}, Y_i's should be chosen such that as many of them (for $i > 0$) have cardinality greater than zero as possible.

The results above can be used to add more axioms until either the data structure becomes 1-correctable or no further addition is possible. In the latter case, a new attribute must be added. This attribute must be generic whenever a failure function lacks a generic attribute. The next lemma shows how the axioms should be generated for the new attribute. For notational simplicity, we assume that the faults in the disambiguation set in question can be numbered consecutively.

Lemma 3.3.2 *Let $IFS = \{f_1, ..., f_n\}$, for some $n \leq N$, be a disambiguation set. Then the addition of the new attribute a_{N+1} will disambiguate these faults if there exist n axioms such that the ith axiom contains the attributes a_i, a_{N+1}, and zero or more attributes from the set $a_{n+1}, ..., a_N$.*

Proof: Let S denote the set of original axioms and f_i, f_j be two distinct faults in IFS. Clearly, $h(f_i)$ will include the axiom $g(a_{N+1}, a_j, \ldots)$ but not $g(a_{N+1}, a_i, \ldots)$. Thus f_i and f_j can be distinguished. Also if there are two faults f_m and f_l that were initially distinguishable, the addition of new axioms cannot make them indistinguishable since all axioms are distinct. Finally, the failure function for f_{N+1} (the fault in the new attribute a_{N+1}) is simply S. Since the original set of faults are detectable, S must be different from all other original failure functions and hence from the modified failure functions. □

3.3.1.2 An Example of a Doubly Linked List

We now consider the example of a circular doubly-linked list (DL) to illustrate the ideas developed above. We choose a circular list here so that the header can be treated like any other element. The basic structure has $AT = \{fp, bp\}$ where fp is the forward pointer and bp is the backward pointer. Both attributes are generic. There is only one (2,2)-ANF axiom satisfied by fp and bp

$$S_1 : \forall E \in EL_{DL}[fp(bp(E)) = E \wedge bp(fp(E)) = E] \tag{3.13}$$

Axiom S_1 states that for any two distinct elements E_1 and E_2 in data structure DL, $fp(E_1) = E_2$ iff $bp(E_2) = E_1$. We know this is true for a circular doubly linked list. Hence S_1 is sound. By a similar argument, it follows that S_1 is complete. If fp (or correspondingly bp) of an element E_1 (E_2) points to E_3 instead of E_2 (E_1), then axiom S_1 is violated. Hence the axiom set is sound, complete and proper.

The possible faults are f_1, f_2 where f_1 is improper forward pointer and f_2 is improper backward pointer. The corresponding failure functions are $h(f_1) = \emptyset$ and $h(f_2) = \emptyset$. Since the failure function is an empty set, from our definition of detectability, it follows that this structure is not 1-detectable and hence not 1-correctable.

The only disambiguation set in the data structure DL is $\{f_1, f_2\}$. We cannot add any more axioms since the disambiguation set includes all the attributes of the basic structure. Thus we need to add a new attribute, say Q, for disambiguation. Two new axioms can be generated using Q, one each with fp and bp. Q must be such that its value is affected by changes in one or more instances of either fp or bp. The count of the number of elements (including the header) in the list appears to be such an attribute (although the later discussion on internal compensation will show that count remains unaffected by changes in multiple instances of fp and bp). Now we obtain a new data structure CL with $AT = \{fp, bp, count\}$, and $AX = \{S_1, S_2, S_3\}$ where S_1 is same as above and

$$S_2 : \forall E \in EL_{CL}\,[fp^{count}(E) = E] \tag{3.14}$$

$$S_3 : \forall E \in EL_{CL}\,[bp^{count}(E) = E] \tag{3.15}$$

Axiom S_2 (S_3) states that, starting from any element we can traverse through the entire data structure (including the header) if we follow the pointer fp (bp) *count* times. The axiom set is sound, but not proper or complete. The problem is that both new axioms continue to hold even if we multiply the value of *count* by an integer, which amounts to saying that we return to the starting element if we traverse the entire data structure more than once. Thus both S_2 and S_3 are improper. To see the incompleteness of AX, note that the data structure D possesses the following properties:

$$C_1 : \forall E\, \forall n\, 1 \leq n < count\,[fp^n(E) \neq E] \tag{3.16}$$

$$C_2 : \forall E\, \forall n\, 1 \leq n < count\,[bp^n(E) \neq E] \tag{3.17}$$

but these properties are not derivable from the three axioms. The obvious choice is to strengthen the terms in S_2 and S_3 by the addition of C_1 and C_2 respectively. However, from a practical viewpoint, it is probably easier to maintain C_1 and C_2 implicitly; i.e., when checking the validity of S_2 or S_3, we make sure that no node is visited more than once. We will therefore ignore C_1 and C_2 from further discussion. With this, and the assumption of no internal compensations, all of our axioms become proper. Now, the base fault set in our example is $F_1 = \{f_1, f_2, f_3\}$ where f_1 and f_2 are as defined previously and f_3 is fault in *count*. The failure functions are:

$$h(f_1) = \{S_3\}$$
$$h(f_2) = \{S_2\}$$
$$h(f_3) = \{S_1\}$$

Thus faults can be identified uniquely i. e. , if only S_3 remains valid, we recognize an instance of data structure with faulty fp, etc. Since every failure function contains an axiom

that uses a generic attribute, we can correct the structure as well. For example suppose that only S_3 remains valid. We first identify the faulty attribute which in this case is fp. Axiom S_3 contains the generic attribute bp. Using this attribute bp and axiom S_1 we correct the data structure as follows. Starting from the header traverse the data structure via pointer bp. At every step of the traversal if $bp(E_1) = E_2$ then assign the address of E_2 to the pointer fp in E_1.

3.3.1.3 Characterization of 1-Correctability

Although we have related 1-correctable structures to disambiguation sets we still have no insight regarding the design requirements that result in a disambiguation set having exactly one member. This will form the topic of discussion in the subsection. Suppose that we have chosen an appropriate value for N, the number of attributes (the choice of an appropriate value for N is discussed in the next section). We show that if the (q,k)-ANF axioms are generated in a particular way, the data structure can be made 1-correctable. We state the following lemma.

Lemma 3.3.3 *To obtain 1-correctability, no attribute need be covered by more than two axioms.*

Proof: From the definitions of detectability and correctability, it is clear that every attribute has to be covered by at least one axiom. Wlog let a_1, a_2,...,a_k be the set of attributes covered by the same axiom. Now to distinguish between the faults in these attributes, exactly $k-1$ of them must be covered by at least one other axiom, such that each attribute has a distinct covering. Let a_i be one of these attributes that is covered by more than two axioms. Let us assume that the attributes are covered by the minimum number of axioms required for 1-correctability and denote the number of axioms covering attribute a_i as δ_i. We show that by replacing one of the axioms containing a_i, say S_q, by another axiom S'_q, we can reduce δ_i and yet maintain 1-correctability.

Obtain S'_q from S_q as follows: $\Gamma(S'_q) = \Gamma(S_q) - \{a_i\} \cup \{a_j\}$ where $\delta_j = 1$ and for all axioms $S_p \in AX, p \neq q, \Gamma(S_p) \neq \Gamma(S_q)$, i. e. the new axiom should not be a replication of an existing axiom. We show that D is 1-correctable with the new set of axioms. Consider a fault in a_i. The new failure function $h'(f_i)$ will contain an additional axiom S'_q. Since the fault f_i was already distinguishable, the addition of S'_q will not affect its distinguishability. The failure functions of faults in attributes common to axioms S_q and S'_q remain the same and hence these faults are distinguishable. The new failure function $h'(f_j)$ no longer contains S_q. Distinguishability between fault f_j and fault, say f_r, in an attribute not common to axioms S_q and S'_q is maintained due to the addition of S'_q in the failure function $h(f_r)$. Distinguishability between fault f_j and faults in attributes common to axioms S_q and S'_q is also retained due to the second axiom covering these attributes. Since the structure was already 1-correctable the second axiom covering these attributes still provides distinguishability.

Repeated use of this construction results in every attribute being covered by at most two axioms since in each step, some attribute, say a_k, with $\delta_k > 2$ will have its δ_k reduced by 1. To complete the proof, we need to show that the number of attributes with $\delta > 2$ is less than or equal to the number of attributes with $\delta = 1$. This is indeed the case, else, repeated application of the construction results in a situation in which every attribute has $\delta \geq 2$ and the structure is 1-correctable. The axioms covering attributes with $\delta > 2$ are superfluous, which contradicts the assumption that the attributes are covered by the minimum number of axioms required for 1-correctability. □

Lemma 3.3.3 gives us the connectivity requirements of the attributes for achieving one correctability. We translate this requirement into the number of axioms required and provide a method of generating them.

Let the number of axioms required be p. So let p distinct attributes be covered by the p axioms exactly once. The remaining $N - p$ attributes must be covered by the same p axioms such that no axiom covers more than k attributes. Each axiom covers k attributes and one attribute in each axiom is covered by exactly one axiom, the $k - 1$ other attributes in an

axiom need to be covered by a second axiom. Since each axiom is to be covered by at most two axioms and the second axiom should produce a distinct covering, the data structure D is 1-correctable if

$$p + p \times \frac{k-1}{2} \geq N \tag{3.18}$$

We can thus determine p, the minimum number of axioms, for a given N and k. We now need to identify the attributes that constitute an axiom. This is accomplished by considering the attribute identification as a graph theoretic problem. The nodes of the graph G_I correspond to axioms and edges correspond to attributes. An edge labeled x joining nodes w and y signifies that the attribute x is covered by axioms w and y. We obtain the attributes constituting an axiom by adopting the following procedure. Before we describe the procedure we need one more definition which we give below.

Definition 3.3.1 *We define the* **distance** *between vertices a_i and a_j of a graph as $min[abs(i-j), abs(N - abs(i-j))]$ where abs gives the absolute value. We denote this as $dist(a_i, a_j)$.*

Procedure 1:

1. Find the smallest value of p that satisfies eqn. 3.18. Draw a graph with p nodes.
2. Generate self loops (edge from a node to itself) at each node. These edges correspond to attributes covered by one axiom only.
3. Generate edges between vertices distance j apart, where $1 \leq j \leq \lfloor \frac{k-1}{2} \rfloor$.
4. If $k-1$ is odd, add edge between vertices distance $\lfloor \frac{p}{2} \rfloor$ apart.
5. Label any N edges with distinct attribute names.

This method of edge generation satisfies the connectivity requirements given in lemma 3.3.3. Each axiom comprises of attributes corresponding to the edges incident on the node corresponding to the axiom. We can now state a property about our method of axiom generation.

Lemma 3.3.4 *If the axioms are generated as in procedure 1, D is 1-correctable. Further, no other method of axiom generation for a 1-correctable structure uses fewer axioms than that used by procedure 1.*

Proof: For the first part, assume that the axioms are generated as outlined in procedure 1. First consider those attributes that are covered by only one axiom. Each one of these attributes belongs to a unique axiom. Hence these faults are mutually distinguishable.

Consider attributes covered by more than one axiom. For any two such attributes occurring in the same axiom, distance 1 edges guarantee that the second axiom in which these attributes occur, is distinct. Thus distinguishability among faults in attributes occurring in two axioms is achieved.

Distinguishability among faults in attributes occurring in one axiom and two axioms is assured since the number of axioms violated in the two cases is different. Thus the structure is 1-correctable.

For the second part, assume that the data structure is 1-correctable. From lemma 3.3.3, it follows that it is sufficient to cover an attribute by at most two axioms. We can thus label the attribute by the axioms that cover it. Let the number of axioms required be p. Since the structure is 1-correctable the attributes should have distinct labels. The number of distinct labels that can be generated with p axioms such that each axiom contains exactly k attributes is given by the equation

$$p + p \times \frac{(k-1)}{2} \tag{3.19}$$

Since procedure 1 uses the smallest value of p satisfying $p + p \times \frac{(k-1)}{2} \geq N$, no other method that achieves 1-correctability can use fewer axioms. □

We now provide an alternate characterization of 1-correctability in terms of the hypergraph representation when the axioms are in (2,2)-ANF. Such a characterization is useful for purposes of analysis.

Lemma 3.3.5 *If the axioms are in (2,2)-ANF, D is 1-correctable if and only if every connected component of G_D has at least 3 nodes and at least 2 nodes have degree ≥ 2.*

Proof: For the only if part, consider a connected component with only 2 nodes a_i and a_j. Of course, this component has only one edge, the one connecting a_i and a_j. It is clear that the fault in a_i cannot be distinguished from that in a_j. Also, if only one node has degree > 2, the removal of this node will make G null. For the if part, let a_i and a_j be two nodes in a connected component with $n \geq 3$ nodes. Let E_1 be one of the edges connecting a_i and E_2 an edge connecting a_j. If E_1 and E_2 are distinct, clearly, the removal of a_i leaves only E_2 and the removal of a_j leaves only E_1. Thus the two faults are distinguishable. Now if $E_1 = E_2$, we must have at least one more edge, denoted E, connecting a_i or a_j. Without loss of generality, we can assume that E connects a_i. Then the removal of a_j leaves behind E but the removal of a_i does not. So again, a_i and a_j are distinguishable. Furthermore, since there is at least one more node (besides a_i) with degree ≥ 2, the removal of a_i does not make G_D null. Thus D is 1-correctable. □

Note: It is desirable to keep the connectivity as uniform as possible. For this reason, we will henceforth assume that for achieving 1-correctability, the axioms are generated such that no node of G_D has degree larger than 2. It is easy to see that this restriction does not require any more axioms than specified by lemma 3.3.5.

3.3.2 Characterization of m-correctable Data Structures

We now turn our attention to the general case of m-correctable data structures when $m \geq 2$. We start by characterizing the minimum number of attributes required for achieving desired level of correctability.

Lemma 3.3.6 *m-correctability requires at least $N = m + k$ attributes. Furthermore, if $N = m + k$, all ${}^{N}C_k$ hyperedges in AG (or (q,k)-ANF axioms) are necessary and sufficient for m correctability.*

Proof: For the first part note that with $m > N - k$, removal of m nodes will make the fault graph empty. For $m = N - k$, the removal leaves precisely one hyperedge, moreover, this hyperedge is unique for every distinct m-ary fault. Thus all m-ary faults can be

distinguished. Now if we consider a subgraph of G by leaving out one of the N nodes (say a_i), the same argument can be applied again to conclude that all $(m-1)$-ary faults in $FS-f_i$ can be distinguished. Now since the node a_i will contribute at least one new hyperedge for $(m-1)$-ary faults, all these faults must be distinguishable from all m-ary faults. By induction, it follows that faults of any arity in the range $1..m$ must be distinguishable. This proves sufficiency. □

Intuitively, a fault graph is empty if the number of vertices corresponding to non-faulty attributes in the fault graph is less than k. This is due to the fact that every hyperedge contains k attributes. If the number of vertices, corresponding to non-faulty attributes, is exactly k then the fault graph will contain one edge if the hypergraph representing the data structure was a complete hypergraph. Moreover, for every distinct m-ary fault, the fault graph will contain a distinct hyperedge and hence the structure is correctable.

3.3.2.1 Design using Minimum Number of Attributes

Consider the design of a 2-correctable data structure TC starting with the 1-correctable data structure CL designed in section 3.3.1.2. Let us call the attributes *fp*, *bp* and *count* as a_1, a_2 and a_3 respectively. Obviously, any double fault, f, makes $G_{CL}(f)$ null. Since we have generated all possible ANF axioms using the attributes of CL, we must first add another attribute, say a_4. Attribute a_4 must be generic since we need 3 generic attributes for 2-correctability. Since any double fault in the data structure TC leaves 2 attributes non-faulty and 2-detectability requires the existence of an edge connecting the vertices corresponding to the non-faulty attributes, G_{TC} should be a complete graph and is as shown in fig. 3.1.

Thus we need axioms (a_1, a_4), (a_2, a_4), and (a_3, a_4) in data structure TC. For the data structure TC to be 2-correctable we require that every double fault produce a distinct fault graph. Consider two distinct double faults f_a and f_b in TC. Since G_{TC} is a complete graph, $G_{TC}(f_a)$ and $G_{TC}(f_b)$ will be different and hence data structure TC is 2-correctable. The fault graph of a single fault is distinct from the fault graph of a double fault since the

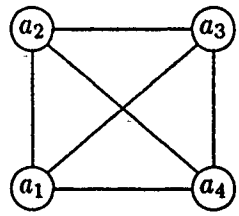

Figure 3.1: Graph model of data structure TC.

number of edges present in the fault graph of a single fault is more than the number of edges present in the fault graph of a double fault. Viewed in terms of failure functions, the failure functions for single faults are:

$$h(f_1) = \{(a_3,a_4), (a_3,a_2), (a_4,a_2)\} \quad h(f_2) = \{(a_3,a_4), (a_3,a_1), (a_4,a_1)\}$$
$$h(f_3) = \{(a_1,a_2), (a_1,a_4), (a_2,a_4)\} \quad h(f_4) = \{(a_1,a_2), (a_1,a_3), (a_2,a_3)\}$$

and those for double faults are:

$$h(f_1,f_2) = \{(a_4,a_3)\} \quad h(f_1,f_3) = \{(a_2,a_4)\} \quad h(f_1,f_4) = \{(a_2,a_3)\}$$
$$h(f_2,f_3) = \{(a_1,a_4)\} \quad h(f_2,f_4) = \{(a_1,a_3)\} \quad h(f_3,f_4) = \{(a_1,a_2)\}$$

Note that all failure functions are distinct (as expected, since the fault graphs are distinct) and have an axiom involving a generic attribute. Thus the structure is 2-correctable.

One possible choice for attribute a_4 is the alternate pointer (ap). An alternate pointer is a generic attribute and points to the second successive element. We assume that the data structure TC contains odd number of elements and two headers (the necessity of this assumption is given below). With this choice of a_4 the data structure TC has attributes $AT = \{fp, bp, count, ap\}$ and axioms $AX = \{S_1, \ldots, S_6\}$ where $S_1, \ldots, S_3$ are the same as in CL and $S_4, \ldots, S_6$ are given below.

$$S_4 : \forall E \in EL_{TC}\ [ap(E) = fp^2(E)] \tag{3.20}$$

$$S_5 : \forall E \in EL_{TC} \exists E' \in EL_{TC} :\ [ap(E) = E' \Leftrightarrow bp^2(E') = E] \tag{3.21}$$

$$S_6 : \forall E \in EL_{TC}\, [ap^{count}(E) = E] \tag{3.22}$$

Axiom S_4 states that if element E_2 is pointed to by ap in element E_1, then there is an element, say E_3, belonging to the data structure such that pointer fp in E_3 points to E_2 and fp in E_1 points to E_3. S_5 is similar to S_4 but relates 3 elements via attribute bp. Axiom S_6 is similar to axioms S_2 and S_3. As such axiom S_6 is improper and the axiom set is incomplete since data structure possesses the property

$$C_3 : \forall E \forall n\, 2 \leq n < count\, [ap^n(E) \neq E] \tag{3.23}$$

which is not derivable from AX. The axiom set can be made complete by strengthening the term of S_6 by adding C_3 to it. We ignore C_3 with the understanding that it is maintained implicitly. We make note of few implementation details that we need to contend with.

- Because ap points to the next successive element we need two headers. In the absence of two headers the alternate pointer points to itself when the data structure has no elements and when it has exactly one element. We refer to the two headers as the primary and secondary headers with the primary header being the first element. It is sufficient if only the primary header remains accessible externally.

- We require that there always be an odd number of elements in the data structure (including the headers). This can be accomplished by adding a dummy element when there are even number of elements. With an even number of elements we may be able to recover only half the elements if both fp and bp are faulty. This is because ap points only to alternate elements and hence connects the data structure via two mutually distinct chains. We can avoid the additional dummy element if both the headers are externally accessible but the axioms will be different in this case.

- In case fp or bp are non-faulty then the correction mechanism is similar to that of data structure CL. In case ap remains non-faulty then we first traverse through the entire data structure obtaining all the elements. Since we know the relative positions of the

primary and secondary headers the position of the other elements can be ascertained. We can now correct fp and bp by traversing the data structure starting at the primary header.

3.3.2.2 Design using Minimum Number of Axioms

It may not be always possible to obtain attributes and axioms such that the graph representing the data structure is complete. Thus, it is interesting to explore m-correctability requirements when $N > m + k$.

Lemma 3.3.7 *For axioms in (q, k)-ANF m-connectivity of AG is necessary for m-correctability.*

Proof: By contradiction. Consider a node a_i that occurs in p hyperedges. Let $\eta_i = \{x_1, ..., x_p\}$ where $x_j, j \in [1..p]$ denotes an attribute contained in the jth hyperedge involving a_i but in no other. Suppose that $p = m - 1$. Now consider the $(m-1)$-ary fault g defined by $\omega(\eta_i)$. Since the removal of all nodes in η_i will also remove a_i, g cannot be distinguished from the m-ary fault $g \cup f_i$, and we have the contradiction. □

For $k = 2$, every edge has a unique distance associated with it. The edge distance is the distance between the two vertices comprising the edge. We now show that if the axioms are generated in a particular way, m-connectivity is also sufficient for m-correctability.

Lemma 3.3.8 *If $k = 2$ and $2 \leq m \leq N - 3$, m-connectivity is sufficient for distinguishing between all m-ary faults provided that we have all edges of distance up to $\frac{m}{2}$, and if m is odd, at least one additional edge of distance $\frac{m+1}{2}$ per node.*

Proof: First, we argue that if the lemma is true for $m = N - 3$, it must also be true for $m < N - 3$. This follows from the simple observation that with fixed connectivity, more nodes in G only result in additional (but noncompensating) arcs in the presence of any fault. Thus more nodes cannot affect distinguishability.

We now assume that $m = N - 3$ and show that all m-ary faults can be detected and are distinguishable. With $m = N - 3$, there are only three nonfaulty attributes, henceforth

denoted as a, b and c. Now to prove detectability, we must show that for any $f \in F_m$, $G(f)$ must contain at least one of the three possible arcs between these nodes. Since G has all arcs of distance up to $\frac{m}{2}$, and if m is odd, at least one arc of distance $\frac{m+1}{2}$, the only way to avoid having arcs ab and ac is to have $dist(a,b) \geq \frac{m}{2}+1$ and $dist(a,c) = \frac{m+1}{2}+1$[1]. However, this would imply that $dist(b,c) \leq 1$. Thus $G(f)$ cannot be empty. Now to prove that $G(f)$ must be unique, let g_1 and g_2 be two m-ary faults. Denote the non-faulty attributes of g_1 and g_2 as a,b,c and a',b',c' respectively. Since g_1 and g_2 are distinct, $a,b,c \neq a',b',c'$. Thus there are three cases to consider:

1. $a \neq a', b \neq b', c \neq c'$. From the detectability proof, we know that $G(g_1)$ must contain at least one of the three arcs between a,b,c. Similarly, $G(g_2)$ must contain at least one of the three arcs between a',b',c'. Thus $G(g_1)$ and $G(g_2)$ must be distinct.

2. $a = a', b \neq b', c \neq c'$. Since every edge involves at least two nodes, by the same argument as in case (1), edge(s) in $G(g_1)$ should still be different from $G(g_2)$.

3. $a = a', b = b', c \neq c'$. Here the only troublesome situation is the one where both $G(g_1)$ and $G(g_2)$ include only the edge ab. Now for ac and bc to be absent from $G(g_1)$, the distance between a and b cannot exceed 1 (as already shown above). Now, to avoid arc ac' from $G(g_2)$, c' must be located at a distance $\frac{m+1}{2}+1$ from a (see fig 3.2). But this implies that the distance between b and c' must be less than $\frac{m+1}{2}$, and therefore, bc' must belong to $G(g_2)$.

Thus any two faults are distinguishable. □

When m is odd, our method requires an additional edge of distance $\frac{m+1}{2}$ per node. This may, in some cases, necessitate more than the minimum number of edges, since it may not be possible to generate edges that satisfy both the distance and connectivity requirements. Figure 3.3 shows an example of a 3-correctable data structure requiring ten edges. However,

[1] Another possibility is to have $dist(a,b) \geq \frac{m+1}{2}+1$ and $dist(a,c) = \frac{m}{2}+1$, but because of symmetry, it can be ignored.

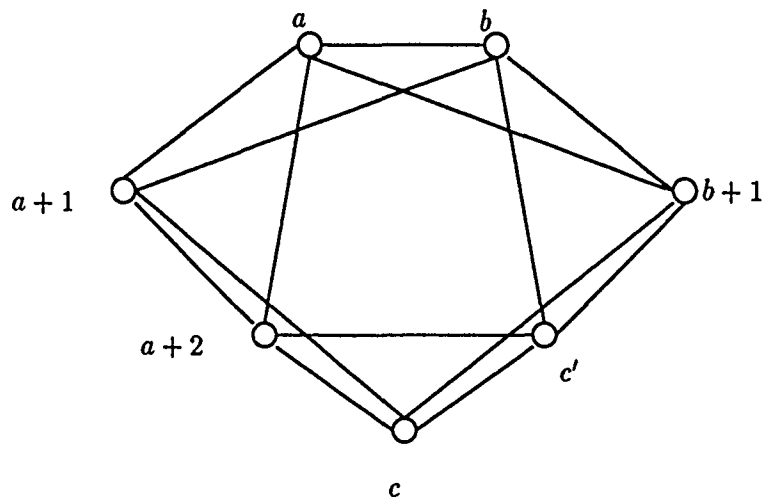

Figure 3.2: A 4-correctable data structure with 7 attributes.

if we relax the distance requirement we can do better. The modification is as follows: the additional edge should have distance greater than $\frac{m}{2}$. With this change, we obtain a 3-correctable data structure with nine edges as shown in figure 3.4. Note that two of the additional edges are of distance $\frac{m+1}{2}$ and one edge of distance $\frac{m+3}{2}$. We do not use this scheme in the rest of the thesis since a method to generate edges of varying distances is not known.

Lemma 3.3.9 *If $k = 2$, G_D is m-connected ($m \geq 2$) and all m-ary faults are distinguishable, then D is m-correctable.*

Proof: We need to show that all p-ary faults for $1 \leq p \leq m$ are distinguishable. This is done by induction on p. For the basis, we choose $p = m$, since by assumption all m-ary faults are distinguishable. For the inductive step, suppose that all faults with arity in the range $p + 1..m$ are distinguishable. Since all edges of distance 1 are present, it is easy to see that all single faults are distinguishable. Let g be a p-ary fault and f some single fault such that $\omega^{-1}(f) \cap \omega^{-1}(g) = \emptyset$. Let fg denote the $(p+1)$-ary fault $f \cup g$. Obviously, $h(fg) = h(f) \cap h(g)$. We need to show the following:

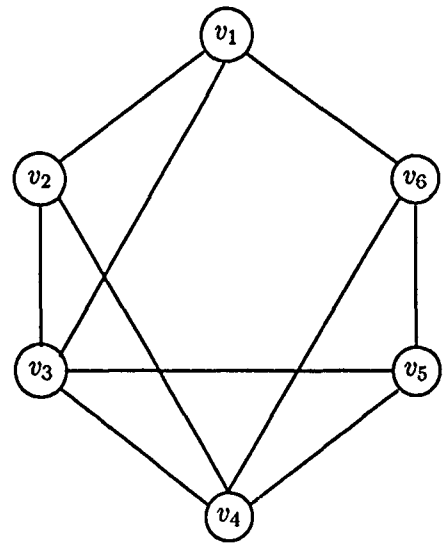

Figure 3.3: A 3-correctable data structure with ten edges.

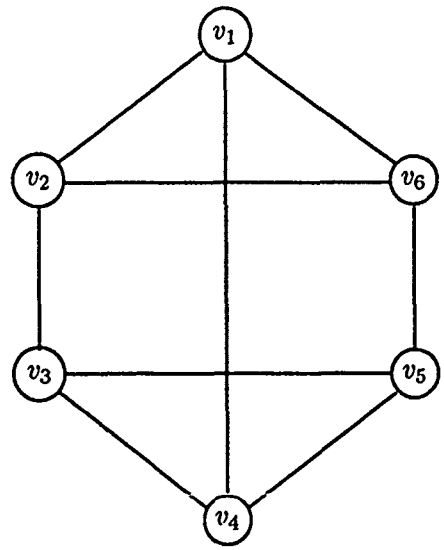

Figure 3.4: A 3-correctable data structure with nine edges.

1. All p-ary faults are distinguishable. For this, let g' be another p-ary fault and consider two $(p+1)$-ary faults fg and fg' where f is a single fault. By our assumption, $h(fg) \neq h(fg')$ that implies that $h(f) \cap h(g) \neq h(f) \cap h(g')$. Therefore, $h(g) \neq h(g')$.

2. $h(fg) \neq h(g)$, i.e., p-ary faults cannot be confused with $p+1$-ary faults. We show this by contradiction. That is, let $h(fg) = h(g)$. This implies that $h(g) \subset h(f)$. Thus all the attributes that are neighbours of $\omega^{-1}(f)$ must be included in $\omega^{-1}(g)$. But according to our assumption $\omega^{-1}(f)$ must have at least m neighbours, and we have a contradiction.

3. p-ary faults are distinct from $(p+i)$-ary faults, for $i > 1$. This follows from induction and the following fact: if f and f' are two distinct single faults and g some other fault then from (2) we know that $h(f'fg) \subset h(fg) \subset h(g)$.

Since all faults are detectable, the faulty attributes can be identified. Thus the entire data structure can be corrected as there are at least $m+1$ generic attributes. □

Lemmas 3.3.8 and 3.3.9 together show the desired sufficiency condition for m-correctability. Our results do not take into account the practical considerations of whether the required axioms can indeed be generated. In this regard, ordering the attributes in a particular way and/or increasing the number of attributes are the two options available to a designer.

Consider the design of a 2-correctable data structure TM. Let the attributes of TM be $a_1, \ldots, a_5$. Since TM has 5 vertices, any double fault leaves 3 non-faulty attributes. Detectability of the fault requires the existence of an edge between any two of the three non-faulty attributes. Correctability requires that every distinct fault (single or double) results in a distinct fault graph.

To obtain the graph G_{TM}, we use lemma 3.3.8 and generate all edges between vertices that are distance 1 apart. The resulting graph G_{TM} is shown in fig. 3.5. As desired, each vertex in G_{TM} has connectivity of 2. The data structure TM is given by $AT = \{a_1, \ldots, a_5\}$ and $AX = \{S_1, \ldots, S_5\}$ where $S_i = (a_i, a_{i+1}), 1 \leq i \leq 4$ and $S_5 = (a_5, a_1)$.

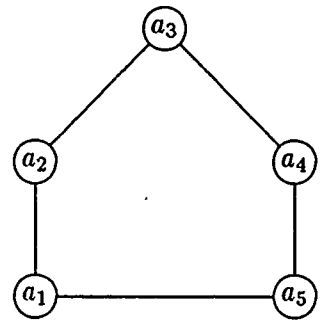

Figure 3.5: Graph model of data structure TM.

3.3.3 Faults with Compensation

As stated in the previous chapter, compensation refers to the situation where the effect of one fault is nullified by another, thereby retaining the validity of some of the axioms containing the faulty attributes. We also introduced the notions of external and internal compensations. This distinction becomes necessary because of our use of generic fault model, where one or more faults in the individual instances of a generic attribute (say, i-faults) are treated as a single fault in the generic attribute (say, a-fault or just fault). Thus external compensation refers to the compensation between distinct a-faults, and internal compensation between different i-faults corresponding to the same a-fault.

3.3.3.1 Internal Compensations

Consider faults in multiple instances of the same generic attribute a. Let the faults compensate and result in a faulty data structure instance that is a permutation of the non-faulty data structure instance (i. e. the set of pointer values is the same but not the ordering). Further, assume that an axiom, say S, involving a generic attribute a and some other non-faulty attribute, say b, remains valid. Then S is not proper and to avoid the compensation, we need to make it proper. The evaluation of axiom S requires the traversal of the entire faulty data structure with the aid of generic attribute a. If S is to be proper,

it must distinguish all the correct orderings from the incorrect ones. In particular, if any of the generic attributes imposes a total ordering on the elements, there is only one correct ordering. Therefore, the axiom must be able to capture this ordering. If in the axiom S, attribute b is generic then it is easy to capture the element ordering since there is an instance of b in every element. If b is atomic then S cannot capture this ordering by itself and yet another attribute and term are needed.

A weak generic attribute can be used to enforce an ordering on the elements. But we still need a generic attribute to access an element from another element. Thus we need to add a term relating the generic attribute and the weak generic attribute. The ordering imposed by the weak generic attribute should supplement the ordering (maybe partial) imposed by the generic attribute. An axiom that is insensitive to changes in ordering as a result of a fault in a generic attribute will now be invalidated due to the presence of the weak generic attribute. This makes the axiom proper. However, a term involving a weak generic attribute and generic attribute is not proper since a subset of elements which satisfy the ordering also satisfy the term. Similarly, a term involving a weak generic attribute and an atomic attribute cannot be evaluated since we need to traverse the entire data structure which is not possible without the aid of a generic attribute. Thus every axiom which contains a weak generic attribute should also contain an atomic attribute and a generic attribute. But with this modification the axioms are in (2,3)-ANF and hence our previous design results are no longer applicable. We overcome this drawback by adopting the following design methodology.

We design the data structure using generic attributes and atomic attributes only and assuming no compensations occur (as discussed until now). We then identify the set of axioms that can be affected by internal compensation. As stated earlier these axioms relate a generic attribute and an atomic attribute. For every atomic attribute, say a, present in an axiom belonging to the set identified above, we do the following: Obtain the set, say Q, of generic attributes that are related to attribute a. Add a term relating the weak generic attribute and a generic attribute, say $p \in Q$, to the axiom relating attributes a and p.

We clarify our design methodology by an example. Consider the example data structure CL discussed earlier. Notice that axiom S_2 remains valid as long as we can traverse the entire data structure via fp, visiting each element exactly once. In other words axiom S_2 is insensitive to changes in the access order of elements accessed via fp. The same could be said for axiom S_3 with reference to bp. It is easy to see that the failure function for faults that cause such order changes in fp (or bp) are different from the ones given earlier.

To make axioms S_2 and S_3 proper, we bring in a new generic attribute called *tag* which explicitly defines the ordering, i.e., if element $E1$ precedes element $E2$, then $tag(E1) < tag(E2)$. We could then strengthen axioms S_2 and S_3 to make them proper. For example, in CL, we add a new term to each of the axioms S_2 and S_3.

$$C_1' : \forall E \in EL_{CL}\,[tag(fp(E)) > tag(E)] \tag{3.24}$$

$$C_2' : \forall E \in EL_{CL}\,[tag(E) > tag(bp(E))] \tag{3.25}$$

This change introduces some new problems, namely that the axiom graph remains the same for a fault in either the weak generic attribute or the atomic attribute belonging to the same axiom. Same axiom fault graph implies a loss of distinguishability between two faults. To maintain distinguishability we use the term graph.

Again, as stated in the previous chapter, the term fault graph for a fault is not unique. For our purposes it is sufficient to consider term fault graphs under single edge removals (or violation of single term). We state the following lemma:

Lemma 3.3.10 *Two distinct faults that result in the same axiom fault graph have distinct term fault graphs with respect to single edge removal requirement.*

Proof: Consider two distinct single faults f_1 and f_2 that result in the same axiom fault graph. This can happen in the following way: f_1 is a fault in the weak generic attribute , say a, and f_2 is a fault in an atomic attribute, say b, and attributes a and b are present in the same set of axioms. Because of the way we add weak generic attributes, this can occur whenever all the m vertices adjacent to the atomic attribute represent generic attributes.

Thus faults f_1 and f_2 result in the violation of the same set of axioms and hence the axiom fault graph is same.

Consider the term fault graphs of faults f_1 and f_2. Fault f_1 in the weak generic attribute a affects the term relating the weak generic attribute and the generic attribute only. Similarly fault f_2 in the atomic attribute b affects the term relating the atomic attribute and generic attribute only. Since the atomic attribute and the weak generic attribute appear in the same set of axioms but in different terms the term fault graphs of the two faults are distinct. Moreover, since both these attributes appear only in one term in an axiom, distinguishability is maintained under the single edge removal requirement. Further, since the weak generic attribute always appears along with an atomic attribute, the term graphs are distinguishable even under multiple faults (i.e. if f_1 is a multiple fault that includes the weak generic attribute and f_2 is a multiple fault that includes the atomic attribute).

We now claim that this is the only case when the axiom graphs of two faults are identical. We do this by transforming every hyperedge of the axiom graph into an edge of an equivalent axiom graph. We know that every hyperedge has a tree structure. We merge the children nodes into one and thus obtain an edge in the equivalent axiom graph joining the root vertex (of the hyperedge) and a vertex corresponding to the children vertices (of the hyperedge). Thus the axiom graph is transformed to an equivalent graph where each edge has cardinality of 2 (i.e. k =2). This is possible since initial design of the data structure consists of atomic and generic attributes only and contains two attributes per axiom. The additional terms involving the weak generic attributes are added for the sole purpose of making the axiom proper. Moreover, because of the design methodology adopted, the weak generic attribute and the atomic attribute appear in the same set of axioms. Since, one of the children in the hyperedge corresponds to an atomic attribute, the node corresponding to the children vertices in the equivalent axiom graph should have a connectivity of m (using the results of the previous subsection) and the edges in the equivalent graph satisfy the distance property. Thus no two faults have identical axiom fault graphs which proves our claim and completes the proof of the lemma. □

Henceforth, we shall ignore internal compensations with the understanding that internal compensations arise due to the axiom not being proper. We also ignore weak generic attributes from further discussion since they are introduced solely for the purposes of overcoming internal compensations. Also all future references to the word compensation should be read as external compensations.

3.3.3.2 External Compensations

If external compensations are allowed, an axiom containing more than one faulty attribute may or may not hold. Consequently, the failure functions, as defined above will no longer be unique. To preserve uniqueness, we need to extend the failure function definition. Before we discuss the extension to the failure function we discuss compensation in greater detail and introduce a few concepts.

Suppose that all axioms are in $(2,k)$-ANF. Let f be a p-ary fault and $COMP(f)$ the set of all possible compensations in f. Since only those faulty attributes that occur in an axiom can compensate, we have:

$$COMP(f) = \bigcup_{i=2}^{min(k,p)} COMP_i(f), COMP_i(f) = \{\gamma_i | \omega^{-1}(\gamma_i) \subseteq \omega^{-1}(f), |\omega^{-1}(\gamma_i)| = i\} \tag{3.26}$$

Since f is a p-ary fault and at least two faults are necessary for compensation to take place, COMP(f) denotes the set of all possible combinations of $min(k,p)$ faulty attributes from fault f. Let $c \in COMP_i(f), 2 \leq i \leq min(k,p)$, be an instance of compensation. Then all axioms that contain all the attributes in $\omega^{-1}(c)$ will remain true (in spite of the fact that these attributes are faulty), i.e. i-ary compensation occurs. We denote the failure function under c as $h_c(f)$. For clarity, we denote the h function under no compensation as $h_0(f)$. Note that:

$$h_c(f) = h_0(f) \cup \{S \in AX | \omega^{-1}(c) \subseteq \Gamma(S)\} \tag{3.27}$$

Let $C(f)$ denote the set of compensations allowed for a fault f. If $C(f) = COMP(f)$ for all f, we say that the compensations are **unrestricted**, otherwise, they are **restricted**.

One important class of restricted compensations is **simple compensations** defined as compensations that can occur in the presence of two base faults. Note that a simple compensation is different from a binary compensation. A binary compensation can occur for faults of arity two or more, whereas a simple compensation can occur only for 2-ary faults. Another important class is **Total Compensations**, defined by $C(f) = COMP_{min(k,p)}(f)$, i.e. faults of arity $\min(k,p)$. We illustrate these by an example.

Let $k = 3$ and consider an axiom S such that $\Gamma(S) = \{a_1, a_2, a_3\}$. Clearly, the only faults relevant for compensation are those with arity of 2 or 3. Let $f^{(1)} = \{f_1, f_2\}, f^{(2)} = \{f_1, f_3\}, f^{(3)} = \{f_2, f_3\}, f^{(4)} = \{f_1, f_2, f_3\}$ Then simple compensation means that S can become true only under the first three faults, but not the fourth one. Binary compensations are possible under all four cases. Total compensation means that S can become true only under $f^{(4)}$. The justification for considering only simple compensations may be construed as follows: the probability that all three attributes simultaneously assume incorrect values that still preserve S, is negligible. The justification for considering only total compensations may be given as follows: If a_1 is nonfaulty but a_2 and a_3 are faulty, then the part of S that relates a_1 to the other the other two attributes must be violated, hence S cannot remain true. It is, of course, up to the designer to see if any of these arguments are tenable for the problem at hand.

The extension to the failure function should now be clear: for every fault f, we need to consider $h_0(f)$ and $h_c(f)$ for every $c \in C(f)$, i. e. a data structure is detectable (correctable) if the conditions stated in the definition of detectability (correctability) holds for the failure functions in the set $h_0 \cup \bigcup_{c \in C(f)} h_c$. Unless stated otherwise, we allow unrestricted compensations to occur, i.e. $C(f) = COMP(f)$.

Let D be a data-structure that is m-correctable without compensations. Let f be a q-ary fault and $g \neq f$ a p-ary fault for some $q, p \leq m$. Obviously, $h_0(f) \neq h_0(g)$. Let $INDIS(f,g)$ denote the assertion that faults f and g become indistinguishable due to some compensation, i.e., there exists a $c \in COMP(f)$ and a $c' \in COMP(g)$ such that

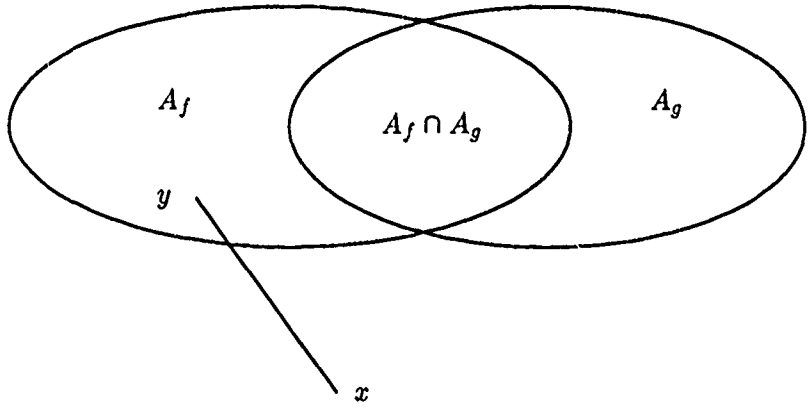

Figure 3.6: Distinguishability of compensating faults

$h_c(f) = h_{c'}(g)$. We now characterize the conditions necessary for $INDIS(f,g)$ to hold by a series of lemmas.

Lemma 3.3.11 *Let $A_f \doteq \omega^{-1}(f)$, $A_g \doteq \omega^{-1}(g)$, and $FG = (A_f - A_g) \cup (A_g - A_f)$. Then $INDIS(f,g)$, if there is no axiom xy such that $x \in AT - (A_f \cup A_g)$ and $y \in FG$.*

Proof: By contradiction. Let xy be such an axiom (see fig 3.6). Without loss of generality, let $y \in A_f - A_g$. Then xy is false for f and cannot be made true by compensation. It is unconditionally true for g. Thus the failure function for fault g must contain xy, with or without compensation. Hence f and g are distinguishable. □

Lemma 3.3.11 states that in order to make two compensating faults indistinguishable every axiom must consist of attributes which belong to either fault f or to fault g.

Lemma 3.3.12 *Let $|A_f - A_g| = n_f$, $|A_f \cap A_g| = n_c$ and $|A_g - A_f| = n_g$. If the axioms are generated as in lemma 3.3.8, then $INDIS(f,g)$ if and only if $n_f + n_c + n_g = N$.*

Proof: By contradiction. Let S_0 denote the set AT - ($A_f \cup A_g$). Let $n_0 = |S_0|$, and by assumption $n_0 > 0$. By lemma 3.3.11, all attributes in S_0 must be at a distance greater than $\frac{m}{2}$ from all attributes in FG. We now show that any numbering of attributes that achieves this will require $n_c \geq m$, a contradiction to the fact that f and g are distinct.

Without loss of generality, we can number the attributes in S_0 as $1..n_0$ (see fig. 3.7). If m is even, then the attributes with numbers $n_0 + 1, \ldots, n_0 + \frac{m}{2}$ and $N - \frac{m}{2} + 1, ..., N$ can

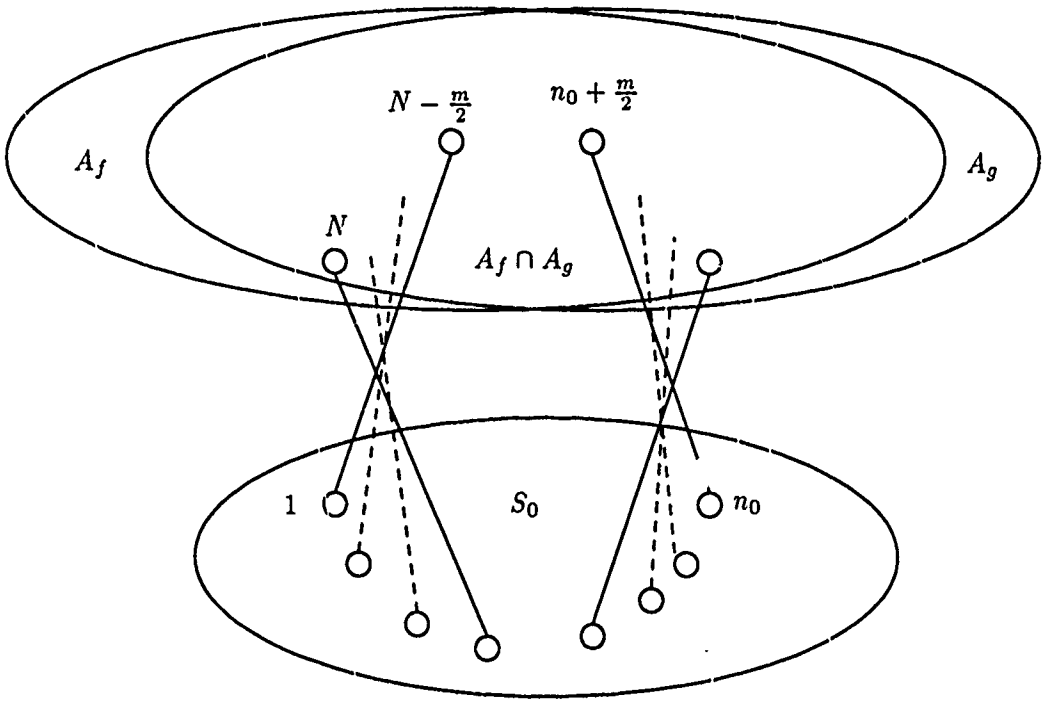

Figure 3.7: Indistinguishability of compensating faults

only occur in $A_f \cap A_g$. If m is odd then the attributes with numbers $n_0 + 1, \ldots, n_0 + \frac{m+1}{2}$ and $N - \frac{m}{2} + 1, \ldots, N$ (or the other symmetric case) can only occur in $A_f \cap A_g$. In either case $n_c \geq m$. Since at most m faults can occur, n_f, n_g is equal to 0, i.e. f is the same as g, a contradiction. □

For example, consider the data structure TC of section 3.3.1.2, and let f_{12} be a compensating double fault in the attributes a_1, a_2 of data structure TC. The failure function of f_{12} is $h(f_1, f_2) = \{(a_1, a_2), (a_3, a_4)\}$. The fault graph $G_{TC}(f_{12})$ is shown in fig. 3.8.

Consider another compensating double fault f_{34} in attributes a_3, a_4 of the same data structure. The failure function of f_{34} is $h(f_3, f_4) = \{(a_1, a_2), (a_3, a_4)\}$. The fault graph $G_{TC}(f_{34})$ is shown in fig. 3.9.

Notice that the two fault graphs are identical and thus the data structure TC is no longer 2-correctable. In the earlier section we had shown that the data structure TC is

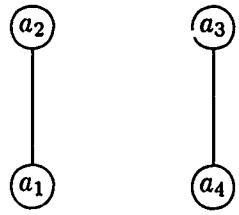

Figure 3.8: Fault graph, $G_{TC}(f_{12})$, of compensating fault f_{12}.

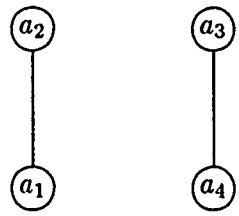

Figure 3.9: Fault graph, $G_{TC}(f_{34})$, of compensating fault f_{34}.

2-correctable if compensations do not occur.

For the data structure TC, assuming unrestricted compensations, the set C (which is the same as $COMP$) is $\{f_{12}, f_{13}, f_{14}, f_{23}, f_{24}, f_{34}\}$. It is easy to see that $h_{12} = h_{34}$, $h_{13} = h_{24}$, $h_{14} = h_{23}$. Thus as stated earlier TC is not 2-correctable in the presence of compensating faults.

Lemma 3.3.12 states that if the axioms are generated using the procedure outlined in lemma 3.3.8 then the two faults become indistinguishable only if they collectively encompass the whole set of attributes. Now we can show the following results:

Corollary 3.3.1 *If $INDIS(f, g)$, both f and g must cover at least two distinct, non-shared attributes, i.e., $n_f \geq 2$, $n_g \geq 2$.*

Proof: Follows from the fact that $p = N - n_f$ and if $n_f < 2, p \geq N - 1$, that is impossible since $m \leq N - 2$. Similarly it can be shown that $n_g \geq 2$. □

Corollary 3.3.2 *If $N > 2m$, all faults remain distinguishable under unrestricted compensations.*

Proof: Let f and g be two faults as in lemma 3.3.12 such that $INDIS(f,g)$ holds. Thus $n_f + n_c + n_g = N$ by lemma 3.3.12. Thus $q + p = N + n_c \geq N$, that further implies that $p \geq N - q \geq N - m$ (since $q \leq m$). Since $p \leq m$, this implies $m \geq p \geq N - m$. Since $N > 2m$, the last inequality becomes $m \geq p \geq m + 1$, which is impossible. Hence f and g are distinguishable. □

The data structure, TM, is 2-correctable even in the presence of compensating faults. Assuming unrestricted compensations the set $C = COMP$ for TM is $\{f_{12}, f_{23}, f_{34}, f_{45}, f_{51}\}$. The failure functions for double faults without compensation (h_0) and with compensation (h_1)[2] in the data structure TM are shown below.

$h_0(a_1a_2) = \{ a_3a_4, a_4a_5\}$
$h_0(a_1a_3) = \{ a_4a_5\}$ $\quad h_0(a_1) = \{ a_2a_3, a_3a_4, a_4a_5 \}$
$h_0(a_1a_4) = \{ a_2a_3\}$ $\quad h_0(a_2) = \{ a_1a_5, a_3a_4, a_4a_5 \}$
$h_0(a_1a_5) = \{ a_2a_3, a_3a_4\}$
$h_0(a_2a_3) = \{ a_4a_5, a_1a_5\}$
$h_0(a_2a_4) = \{ a_1a_5\}$ $\quad h_0(a_3) = \{ a_1a_2, a_1a_5, a_4a_5 \}$
$h_0(a_2a_5) = \{ a_3a_4\}$ $\quad h_0(a_4) = \{ a_1a_2, a_1a_3, a_1a_5 \}$
$h_0(a_3a_4) = \{ a_1a_2, a_1a_5\}$
$h_0(a_3a_5) = \{ a_1a_2\}$ $\quad h_0(a_5) = \{ a_2a_3, a_3a_4, a_1a_2 \}$
$h_0(a_4a_5) = \{ a_1a_2, a_2a_3\}$
$h_1(a_1a_2) = \{ a_3a_4, a_4a_5, a_1a_2\}$
$h_1(a_1a_3) = \{ a_4a_5 \}$
$h_1(a_1a_4) = \{ a_2a_3\}$
$h_1(a_1a_5) = \{ a_2a_3, a_3a_4, a_1a_5\}$
$h_1(a_2a_3) = \{ a_4a_5, a_2a_3, a_1a_5\}$
$h_1(a_2a_4) = \{ a_1a_5 \}$
$h_1(a_2a_5) = \{ a_3a_4 \}$
$h_1(a_3a_4) = \{ a_1a_5, a_1a_2, a_3a_4\}$
$h_1(a_3a_5) = \{ a_1a_2 \}$
$h_1(a_4a_5) = \{ a_1a_2, a_2a_3, a_4a_5\}$

It can be seen that distinguishability is maintained under unrestricted compensation.

[2]We use the subscript 1 for convenience. The subscript 1 should be replaced by the string corresponding to the faulty attributes which compensate, i. e. 1 should be replaced by some c ∈ COMP.

Corollary 3.3.3 *Let $N \leq 2m$ and $S = min(\{|h(f)|, f \in \bigcup_{i=1}^{m} F_i\})$. Then the maximum number of compensations allowed without losing distinguishability is S-1.*

Proof: The size of a failure function is the least for a m-ary fault, so let f be a m-ary fault such that $|h_0(f)| = S$ is minimum. Let g be a fault that compensates with f. Obviously, $|h_0(g)| \geq S$. We have $\omega^{-1}(f) \cup \omega^{-1}(g) = AT$. Thus attributes corresponding to axioms in $h_0(f)$ (denoted $A_{h_0(f)}$) should be in $\omega^{-1}(g)$. Let $h_c(f)$ be equal to $h_c(g)$. This requires that the axioms for attributes belonging to $A_{h_0(f)}$ be regenerated by compensation. If we restrict the number of compensations to S-1 then all the required axioms will not be regenerated, thus maintaining distinguishability. □

The significance of corollary 3.3.3 is that if $N \leq 2m$, but the number of compensations can be restricted to one less than the size of the smallest failure function, we can still maintain distinguishability. We now provide a lower bound on the size of the smallest failure function in terms of N and m. Such a bound is useful for quickly determining if the allowed compensations can be tolerated.

Lemma 3.3.13 *Let $S = min(\{|h_0(f)|, f \in \bigcup_{k=1}^{m} F_k\})$. Then $S \geq \frac{N-m}{2}$. Moreover, the bound is achievable for $m \leq N-4$.*

Proof: First consider the $m = 1$ case. From lemma 3.3.5, the total number of edges in G_D, denoted E, is $\frac{2N+2}{3}$. A single fault will eliminate at most 2 edges (see note at the end of lemma 3.3.5), thus $S \geq \frac{2N+2}{3} - 2$. It is easy to show that this number is no less than $\frac{N-1}{2}$ for $N \geq 4$. Thus we can assume that $m \geq 2$.

Next we show that the bound is achievable for $m \geq N-3$. If $m = N-2$ (the largest possible value), we need a complete graph. Obviously, any m-ary fault will leave only 1 edge, and $\frac{N-m}{2}$ is 1. If $m = N-3$, we can choose the three non-faulty nodes, say a, b and c such that $dist(b,c) = 1$, $dist(a,b) = \frac{m}{2} + 1$ and $dist(b,c) = \frac{m+1}{2} + 1$. The total number of nodes accounted for by this arrangement is $\frac{m}{2} + \frac{m+1}{2} + 3 = m + 3 = N$. Thus we can assume that $N - m \geq 4$ for the rest of the proof.

First suppose that both m and $N-m$ are even. The $N-m$ nonfaulty nodes can be arranged in $\frac{N-m}{2}$ pairs such that the *minimum* distance between any two pairs is maximized. Since $N-m \geq 4$, we must have at least two pairs. Also, since $m \geq 2$, G is connected and has all arcs of distance 1. Consider two pairs containing nodes a, b and c, d respectively. Let δ denote the minimum distance between these two pairs. There are two cases to look at:

1. $\delta > \frac{m}{2}$. Then $dist(a,c) > \frac{m}{2}$ and $dist(b,c) > \frac{m}{2}$, that means that $dist(a,b) \leq 2$. Since $m \geq 2$, it follows that the axiom ab must exist. Similarly, the axiom cd must exist.

2. $\delta \leq \frac{m}{2}$. Using the same argument as in (1), it can be concluded that of the six distances involved, namely ab, ac, ad, bc, bd, and cd, at least two (rather than just one) must be $\leq \frac{m}{2}$. Thus if $dist(a,b) > \delta$ and/or $dist(c,d) > \delta$, we can swap the attributes between pairs such that the members of both pairs have distance $\leq \frac{m}{2}$. Thus axioms must exist within both pairs.

It thus follows that the number of surviving axioms must be at least $\frac{N-m}{2}$. Now if $N-m$ is odd, we can simply leave one attribute aside and conclude that the number of remaining axioms is $\frac{N-m}{2}$. Similar arguments hold when m is odd. The proof shows that the bound is achievable for $m \leq N-3$. In the case of $m = N-4$ the four nonfaulty nodes, say a, b, c, and d can be chosen such that $dist(a,b) = 1$, $dist(c,d) = 1$, $dist(a,c) = \frac{m}{2} + 1$, and $dist(b,d) = \frac{m+1}{2}$. If m is much smaller than N then the bound given in the lemma is a loose bound. □

Although lemma 3.3.13 provides a bound on the minimum number of axioms in the failure function of a compensation free fault, it is not adequate for our purposes. This is because it fails to give information regarding the number of attributes that will be present in the failure function. From lemma 3.3.12, it follows that compensations cause two distinct faults to become indistinguishable only when they encompass the whole set of attributes. By restricting the number of faulty attributes that can compensate to one less than the number of attributes present in the failure function of least size, we overcome indistinguishability of

compensating faults. We will have more to say on this aspect in our discussion on algorithms for identifying faulty attributes.

3.4 Data Structures with More than Two Attributes per Axiom

We now consider the situation when the number of attributes in an axiom is more than two. We assume there is no structure imposed on the hyperedges (unlike the previous case where we assumed a tree structure). We state the requirement on the minimum number of attributes for a certain level of correctability and also provide a method to synthesize a data structure that attains this level of correctability.

Although we had provided the requirement on the minimum number of attributes to achieve m-correctability earlier, this result is not very useful since it necessitates the use of a complete subhypergraphs. We are interested in knowing the number of attributes required to achieve a given level of fault tolerance using the least possible number of hyperedges. We begin our discussion by providing a few additional definitions.

The connectivity τ_i of a vertex, a_i, is defined to be the number of hyperedges which contain a_i. The connectivity τ of the hypergraph is $\min(\tau_i)$. If the cardinality is the same for every hyperedge of the hypergraph then the hypergraph is said to be uniform. A uniform hypergraph of size k has k vertices in every hyperedge.

The definition of distance given earlier needs to be generalized to take into account the fact that there are k attributes per axiom. Given N vertices $a_1, a_2, \ldots, a_N$ the distance between two vertices a_i and a_j, $i > j$, is $dist(a_i, a_j) = \lceil \frac{min((i-j),(j-i+N))}{k-1} \rceil$ where $k > 1$ is the number of vertices in a hyperedge. Note that because of the truncation in the definition of distance, for any given node a_i and distance parameter d, up to $2(k-1)$ nodes will be at distance d from a_i. Also when $k = 2$ we obtain the previous definition of distance.

Definition 3.4.1 *A distance p hyperedge of cardinality k from a vertex a_i contains a_i and $k-1$ contiguous vertices $a_{j_1}, \ldots, a_{j_{k-1}}$ such that* $\text{dist}(a_i, a_{j_l}) = p, 1 \leq l \leq k-1$.

Observation 3.4.1 *For any vertex a_i, the distance of a_i from a distinct vertex a_j i. e. $dist(a_i, a_j)$, is unique.*

We provide an algorithm to construct a hypergraph that is tolerant (correctable and detectable) for a given number of faults. The algorithm given below constructs a hypergraph of N vertices, with k vertices per hyperedge and tolerates up to m (for even m) faults.

Algorithm 2 *Construction of an uniform hypergraph of size k.*

begin
  for $n = 1$ up to $\frac{m}{2}$ **do**
    for each vertex a_i **do**
      begin
        choose $k-1$ contiguous attributes $a_{j_1}, \ldots, a_{j_{k-1}}$
        such that $dist(a_i, a_{j_q}) = n, 1 \leq q \leq k-1$;
        add hyperedge $(a_i, a_{j_1}, \ldots, a_{j_{k-1}})$;
    end
end

Assume that we have enough vertices so that all hyperedges generated by algorithm 1 are distinct.

We shall now show the the hypergraph constructed by algorithm 1 has the desired level of fault tolerance. In the proofs we use the term block to refer to a collection of vertices. Since m-correctability requires m-detectability, we first satisfy this requirement.

Lemma 3.4.1 *The hypergraph constructed by algorithm 1 is m-detectable iff $N \geq m(k-1)+3$.*

Proof: *only if:* Suppose that the hypergraph constructed by the algorithm is m-detectable. This implies that there is at least 1 hyperedge in the fault graph of a fault of arity up to m. We need to show that $N \geq m(k-1)+3$. Assume that $N < m(k-1)+3$. For an hyperedge to be present in the fault graph we need one block of $k-1$, contiguous non-faulty vertices and another non-faulty vertex distant d apart from this block of contiguous vertices, where $1 \leq d \leq \frac{m}{2}$. Let $N \leq m(k-1)$. The N attributes can be considered to be made up of m non overlapping blocks of $k-1$ contiguous attributes. Consider the

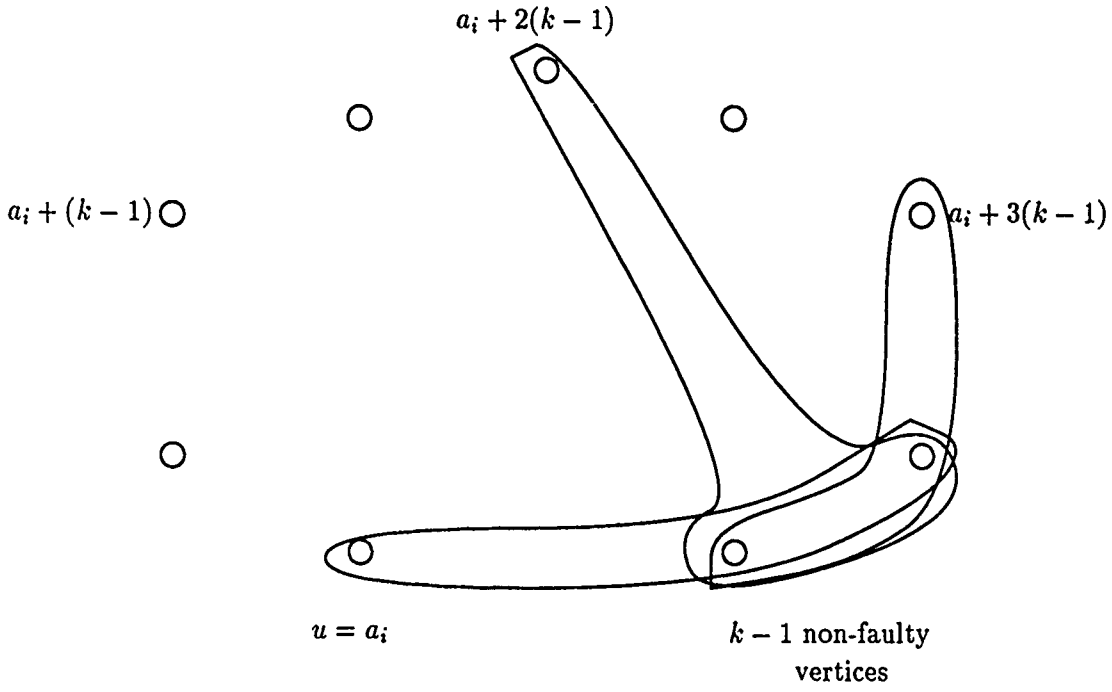

Figure 3.10: A 4-ary fault on a data structure with 9 attributes and $k = 3$.

m-ary fault f_m such that there is one faulty vertex in every block and at most $k-2$ non-faulty vertices between two consecutive faulty attributes. For such a fault the fault graph is empty since we do not have a single block of $k-1$ contiguous non-faulty vertices. Thus $N > m(k-1)$.

With $N = m(k-1)+1$ the vertices can be considered to be made up of m non overlapping blocks of $k-1$ contiguous vertices and 1 block containing a single vertex (say u). Thus any m-ary fault leaves at least one block of $k-1$ contiguous non-faulty vertices. Consider a m-ary fault with faulty vertices $u = a_i, a_{i+(k-1)}, a_{i+2(k-1)}, \cdots, a_{i+(m-1)(k-1)}$ (all subscripts evaluated modulo N). This m-ary fault leaves a block of $k-1$ contiguous non-faulty vertices adjacent to the vertex u. However, all the vertices which have an hyperedge with this block of $k-1$ contiguous non-faulty vertices are faulty (follows from the edge generation method of algorithm 1). Hence the fault graph is empty, so $N > m(k-1)+1$. Fig. 3.10 shows an example of such a m-ary fault for $m = 4$ when $N = 9$ and $k = 3$.

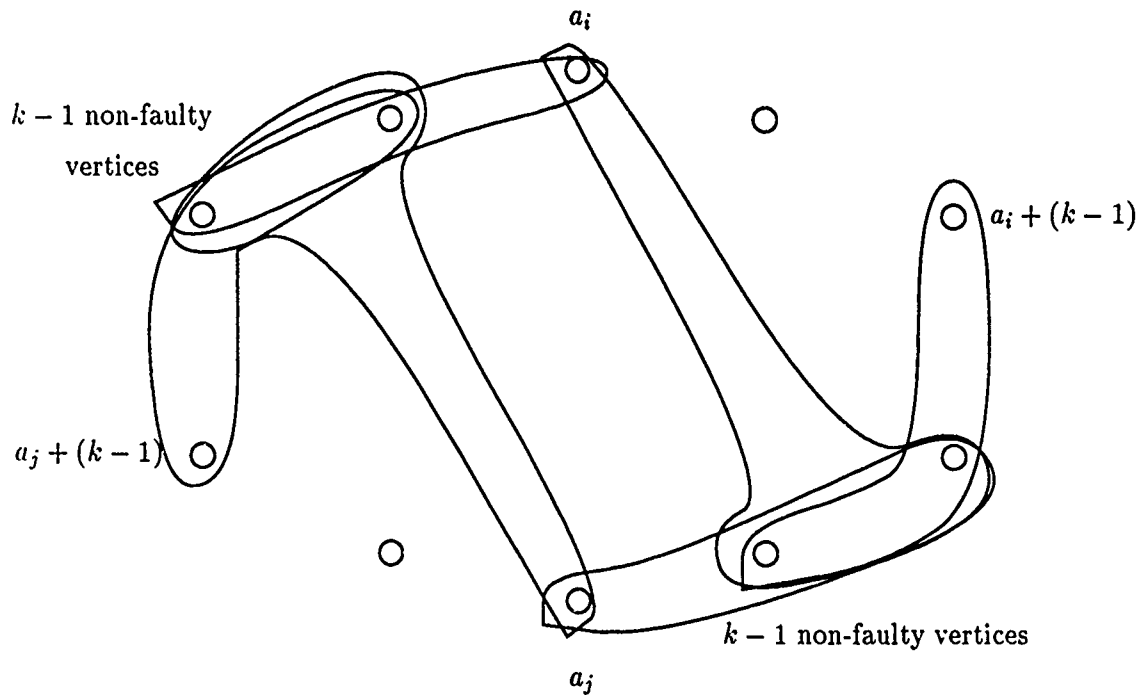

Figure 3.11: A 4-ary fault on a data structure with 10 attributes and $k = 3$.

Let $N = m(k-1)+2$. The N vertices can be considered to be made up of m non overlapping blocks of $k-1$ contiguous vertices and two other vertices, say a_i and a_j. Thus any m-ary fault leaves at least 2 blocks of $k-1$ contiguous non-faulty vertices. Let the blocks be arranged so that there are $\frac{m}{2}$ blocks of $k-1$ contiguous vertices between the vertices a_i and a_j. Consider a m-ary fault f_m with the faulty vertices $a_i, a_{i+(k-1)}, a_{i+2(k-1)}, \ldots, a_{i+(\frac{m}{2}-1)(k-1)}$, $a_j, a_{j+(k-1)}, a_{j+2(k-1)}, \ldots, a_{j+(\frac{m}{2}-1)(k-1)}$ (all subscripts are evaluated modulo N). Thus f_m leaves exactly two blocks containing $k-1$ contiguous non-faulty vertices. Due to the positioning of the two vertices a_i and a_j, faulty vertices which form a hyperedge of distance i with one of the blocks containing $k-1$ contiguous non-faulty vertices form a hyperedge of distance $\frac{m}{2}+1-i$ with the other block containing $k-1$ non-faulty vertices. Since the hyperedges of distance from 1 up to $\frac{m}{2}$ for each of the blocks contains a faulty vertex, the fault graph is empty. Fig. 3.11 shows an example of such an m-ary fault for $m = 4$ when $N = 10$ and $k = 3$.

Thus if $N < m(k-1)+3$ we can find a m-ary fault so that the fault graph is empty.

But this contradicts the fact that the hypergraph is m-detectable. Thus $N \geq m(k-1)+3$ which completes the proof.

if: Let $N \geq m(k-1)+3$. We show that the hypergraph is m-detectable i. e. the fault graph contains at least one hyperedge. With $N \geq m(k-1)+3$, any m-ary fault leaves at least 3 blocks of $k-1$ contiguous non-faulty attributes. Consider one such block, say B_1, of $k-1$ contiguous non-faulty attributes. The block B_1 forms a hyperedge with another vertex located at a distance $d, 1 \leq d \leq \frac{m}{2}$. Since the block B_1 is a member of 2 hyperedges of each distance there are m vertices which are located at a distance of within $\frac{m}{2}$ from B_1 and which form a hyperedge with B_1. If any of these m vertices are non-faulty then the fault graph will be non empty and the proof is complete.

Let all the vertices with which block B_1 forms an hyperedge be faulty. There are $k-2$ non-faulty vertices between a faulty vertex that forms a distance i hyperedge with B_1 and a faulty vertex that forms a distance $i+1$ hyperedge with B_1, $1 \leq i < \frac{m}{2}$. Thus the two other blocks of $k-1$ contiguous non-faulty vertices do not lie within a distance of $\frac{m}{2}$ from block B_1. The total number of vertices within a distance of $\frac{m}{2}$ from block B_1 (including the block B_1) is $(m-2)(k-1)+(k-1)+2$. The $(m-2)(k-1)$ factor is obtained since we can consider the faulty vertex along with the $k-2$ non-faulty vertices between two faulty vertices as one block of $k-1$ contiguous vertices. There are $m-2$ such blocks obtained from $m-2$ faulty vertices. The factor 2 is due to the two other faulty vertices. The factor $k-1$ arises since block B_1 has $k-1$ vertices. Thus the two other $k-1$ blocks must be formed from the vertices that lie at a distance greater than $\frac{m}{2}$ from block B_1. The number of such vertices is $m(k-1)+3-[(m-1)(k-1)+2] = k$. Thus the two other blocks of $k-1$ contiguous non-faulty vertices are formed from this block of k vertices and hence must be overlapping. But since we have a block of k contiguous non-faulty vertices the fault graph will have a distance 1 edge and hence is non empty.

Thus if $N \geq m(k-1)+3$ any m-ary fault leaves a non empty fault graph and hence the fault is detectable. □

We now show that any two faults are distinguishable, provided there are $N \geq m(k-1)+3$ attributes and the axioms are generated as per the algorithm.

Lemma 3.4.2 *If the hypergraph generated consists of $N \geq m(k-1)+3$ nodes, then any two distinct p-ary faults, $p \leq m$, are distinguishable.*

Proof: Any p-ary fault, $p \leq m$, results in the creation of blocks of non-faulty vertices. A block can consist of $k, k-1$ or less than $k-1$ vertices. Two or more blocks containing k vertices may overlap. Similarly a block containing $k-1$ vertices may overlap with another block containing $k-1$ vertices or a block containing k vertices. Since a block containing at least $k-1$ contiguous vertices is necessary for an hyperedge to be present in the hypergraph, we confine our attention to blocks containing $k-1$ contiguous vertices or more.

If the two distinct faults, say q and r, result in different number of blocks containing k contiguous vertices then distinguishability is immediate. This is because each block containing k contiguous vertices corresponds to a distance 1 hyperedge. If the number of blocks containing k contiguous vertices in the two faults q and r are same but the vertices comprising the blocks are different then again distinguishability is obtained.

We are now left with the case when distinct faults q and r result in identical blocks containing k contiguous vertices and identical blocks containing less than $k-1$ contiguous vertices (this is possible only if the faults are of same arity). Since the two faults are different there must be at least one block, say B_q, containing $k-1$ contiguous non-faulty vertices in fault q which is not present in fault r and one block, say B_r, containing $k-1$ contiguous non-faulty vertices in fault r not present in fault q. Let the block B_q contain vertices $q_1, q_2, \ldots, q_{k-1}$ and the block B_r contain vertices $r_1, \ldots, r_{k-1}$. Further let each q_i be different from all r_i and q_{k-1} and r_{k-1} be consecutive attributes. Let fault q correspond to r_{k-1} being faulty and r correspond to q_{k-1} being faulty (the other $m-1$ faulty attributes are common to both the m-ary faults). Assume that the two faults are indistinguishable. Of the two hyperedges of distance $\frac{m}{2}$ using the block B_q, the vertex, say u, closer to q_1 will not have an hyperedge with the block B_r. Similarly the vertex, say v, closer to r_1 will not have

an hyperedge with the block B_q. There are at most $m-2$ vertices which have an hyperedge with both the blocks B_q and B_r. Each of these $m-2$ vertices must be faulty otherwise distinguishability is immediate since the hyperedge formed by any such vertex with B_q is different from the hyperedge formed with B_r. We now have two cases to consider:

- *case i: u is faulty:* In case of fault q all the hyperedges with the block B_q are destroyed. In case of fault r the hyperedge $r_1, \ldots, r_{k-1}, v$ will be present in the fault graph.
- *case ii: v is faulty:* In case of fault q the hyperedge $q_1, \ldots, q_{k-1}, u$ will not be destroyed whereas in case of fault r all hyperedges with the block B_r are destroyed.

Thus in either case the fault graph is different for the two faults q and r, a contradiction to the assumption that the faults are indistinguishable. Thus any two faults are distinguishable, which proves the lemma. □

These two lemmas put together lead us to the final result which is

Corollary 3.4.1 *The hypergraph generated by algorithm 1 is m-correctable if and only if* $N \geq m(k-1)+3$.

It is interesting to note that these results are the generalizations of lemmas 3.3.8 and 3.3.9 for $k > 2$ when m is even. Another interesting observation concerns compensations. With $N > m(k-1)+3$, for all values of $k > 2$ the total number of attributes is greater than $2m$ and from corollary 3.3.2 it follows that compensations do not affect distinguishability of faults.

Chapter 4

Identification of Faulty Attributes

In this chapter we discuss algorithms for identifying faulty attributes in a data structure designed according to the method discussed earlier. Again, we confine our discussion to the situation where each axiom has two attributes. Our problem is as follows: Given an instance of a faulty data structure and the data structure specification, i. e. , the set of attributes and axioms, we need to obtain the set of faulty attributes. From the faulty data structure instance and the data structure specifications we can obtain the failure function. Alternatively, the failure function along with the data structure specification could be provided as input. In this chapter we assume that the latter alternative is used. We start by introducing a few definitions.

We define a mapping $\Gamma : AX \mapsto AT$, where Γ(S) gives the set of attributes present in axiom S. We extend this mapping to a set of axioms, Z, as $\Gamma(Z) = \bigcup_{S \in Z} \Gamma(S)$. We also define a function $\gamma : AT \mapsto AX$, where $\gamma(a)$ is the set of axioms containing the attribute a. For a set of axioms Z we define a subset of axioms restricted to a set P of attributes, denoted Z_P, as $\{S_i | S_i \in \gamma(a), a \in P\} \cap Z$, i. e. it is the set of those axioms of Z which involve at least one attribute from the set P.

4.1 Characteristics of Failure Function

We study the properties of the failure function. The attributes and the axioms present in the failure function display characteristics which are useful in identifying the faulty attributes.

The failure function of any fault contains valid axioms. However, it is possible that certain non-faulty attributes are not constituents of any axiom present in the failure function. The lemma given below provides a bound on the number of non-faulty attributes that are

not constituents of any axiom in the failure function.

Lemma 4.1.1 *For any p-ary fault f, $p < m$, every non-faulty attribute is present in $\Gamma(h(f))$ and for $p = m$, at most one non-faulty attribute is absent from $\Gamma(h(f))$.*

Proof: By lemma 3.3.8, every attribute a has m neighbours. Thus if $p < m$, and a is non-faulty, it must have at least one non-faulty neighbour, which means that a must belong to Γ(h(f)). Thus for $p < m$, no attribute can be absent from Γ(h(f)). Therefore we only need to consider the case $p = m$.

Now for $p = m$, if a is non-faulty and absent from Γ(h(f)), all the neighbours of a must be faulty. Let a', distinct from a, be another non-faulty attribute absent from Γ(h(f)). Then, all the m neighbours of a' should be faulty. But since we have only m faults, a and a' should have the same neighbours. But from the connectivity specifications of lemma 3.3.8, this is impossible. Hence a and a' must be the same. Thus there can be at most one non-faulty attribute absent from Γ(h(f)). □

The next lemma provides a relationship between the non-faulty attributes present in the failure function and the faulty attributes of the data structure.

Lemma 4.1.2 *Every faulty attribute of a p-ary fault f, $p \leq m$, has as its neighbour a non-faulty attribute which belongs to $\Gamma(h(f))$.*

Proof: By lemma 3.3.8, every faulty attribute has a non-faulty neighbour. Now if $p < m$, it follows from lemma 4.1.1 that every non-faulty attribute is present in $\Gamma(h(f))$ and we are done. So let f be a m-ary fault. If $N = m + 2$, the graph must be fully connected, hence every faulty attribute has a neighbour in $\Gamma(h(f))$. Let $N \geq m + 3$. From lemma 4.1.1, at most one non-faulty attribute, say a, is absent from $\Gamma(h(f))$. From the connectivity specifications of lemma 3.3.8, it follows that no two attributes can have the same set of attributes as neighbours. Since a has all the faulty attributes as neighbours, every faulty attribute must have a non-faulty attribute, other than a, as a neighbour. Also, since a is absent from $\Gamma(h(f))$, all the neighbouring non-faulty attributes must be in $\Gamma(h(f))$. This proves the lemma. □

4.2 Faults without Compensation

The failure function of a compensation free fault contains axioms of non-faulty attributes only. Lemma 4.1.2 immediately gives us the algorithm for identifying the faulty attributes. The algorithm first computes the set $X = \Gamma(h(f))$. From lemma 4.1.2 it follows that every faulty attribute has as its neighbour a non-faulty attribute belonging to $\Gamma(h(f))$. Hence the faulty attributes can be obtained by scanning the axioms in $AX - h(f)$ restricted to the attributes in X. The complete algorithm is described below.

Algorithm 3 *Identification of faulty attributes.*

Input: a. Data structure specification.
b. An instance of the data structure under a fault f.
Output:
faulty: set of faulty attributes
nfaulty: set of non-faulty attributes.
begin
valid $\leftarrow h(f)$;
violated $\leftarrow AX - h(f)$;
faulty $\leftarrow$ {}; (* set of faulty attributes *)
nfaulty $\leftarrow \Gamma$(valid);
for each a in nfaulty **do**
faulty $\leftarrow$ faulty $\cup\{b|\Gamma(S) = (a,b), S \in$ violated$\}$;
nfaulty $\leftarrow AT$- faulty;
end .

Lemma 4.2.1 *The algorithm to identify the faulty attributes executes in* $\mathbf{O}(mN)$ *steps.*

Proof: We shall assume that the failure function is given. To obtain the faulty attributes, each axiom in the set *violated* is scanned once. There are N axioms of each distance and axioms of every distance from 1 up to $\frac{m}{2}$. In case m is odd there is an additional axiom for each attribute. Thus the total number of axioms is at most $\frac{N \times m}{2} + N$. Thus the total number of steps is $\mathbf{O}(mN)$, which proves the lemma.

In case the failure function is not given we need an additional $\mathbf{O}(mNM)$ steps since we have $\mathbf{O}(mN)$ axioms, each of which involves examining all M elements of the data structure.

□

4.3 Faults with Compensation

Compensation refers to the situation where the effect of one fault is nullified by another, thereby retaining the validity of some of the axioms containing the faulty attributes.

4.3.1 Internal Compensation

As stated in the previous chapter, internal compensation is a non problem, since its presence merely indicates that the axioms are improper. Internal compensations can be overcome by adding a **weak generic** attribute to an axiom involving an atomic attribute and a generic attribute. Weak generic attributes, like generic attributes have an instance of the attribute in every element. However, given an element of the data structure they cannot be used to access another element in the data structure. Weak generic attributes are used to reinstate certain properties (like element ordering, membership) in an axiom involving an atomic attribute and generic attribute.

With the addition of a weak generic attribute the axioms are no longer in 2-ANF form but are in (2,3)-ANF form. However, since a weak generic attribute is always associated with an atomic attribute, the (2,3)-ANF axioms can be converted into (2,2)-ANF axioms by simply ignoring the weak generic attribute in our hypergraph model. In other words, we coalesce the weak generic attribute into the atomic attribute associated with it. With this transformation, we cannot distinguish between a fault in the weak generic attribute and a fault in the atomic attribute associated with the weak generic attribute. Moreover, the fault will always show up as a fault in the atomic attribute. This drawback can be overcome in the correction procedures as discussed later. Thus, the problem of identifying faulty attributes in the presence of internal compensation reduces to the problem of identifying faulty attributes in a system without compensation, which was discussed in the previous section. Hence, in all further discussions we restrict our attention to the problem of identifying faulty attributes in the presence of external compensations. Also, all future references to the term compensation pertains to external compensation.

4.3.2 External Compensation

If external compensations are allowed, an axiom with both attributes faulty may or may not be violated. Consequently the failure functions will no longer be unique.

Since external compensations cannot occur if only single faults are allowed, we henceforth assume that the data structure is designed for handling at least 2 faults, i. e. $m \geq 2$.

Let $h(f)$ be the failure function as computed from the axioms that are found valid. Then $h(f)$ can be expressed as $h_0(f) \cup X$, where $h_0(f)$ is the failure function without compensations and X is some subset of axioms that result from compensation, i. e. $X \subseteq \{S_i | \Gamma(S_i) \subseteq \omega^{-1}(f)\}$. The fault identification is now more difficult because X is not known a priori. Thus $\Gamma(h(f))$ consists of both faulty and non-faulty attributes. We denote the number of faulty attributes in the failure function as $P(m)$. From corollary 3.3.2, it follows that if $N > 2m$, then $P(m) \leq m$. When $N \leq 2m$, $P(m) \leq N - m - 1$. The reason for $P(m) \leq N - m - 1$ is as follows. Corollary 3.3.3 states that distinguishability among faults is maintained if the number of compensations is limited to one less than the size of the smallest failure function. Since the number of non-faulty attributes is at least $N - m$, the distinguishability will be maintained if we limit the number of faulty attributes regenerated due to compensation to $N - m - 1$ and thus we obtain the value of $P(m)$.

We make another observation which we use frequently in the rest of the thesis.

Lemma 4.3.1 *The number of faulty attributes in the failure function of any fault is less than or equal to the number of non-faulty attributes in that failure function.*

Proof: There are at least $N - m$ non-faulty attributes in a data structure in the presence of any fault of arity up to m. It follows from lemma 4.1.1 that at most one of these non-faulty attributes is absent from the failure function of any m-ary fault. Thus the failure function contains at least $N - m - 1$ non-faulty attributes. If $N > 2m$, then the number of non-faulty attributes in the failure function is $N - m - 1$ which is $\geq m \geq P(m)$. If $N \leq 2m$ then the number of non-faulty attributes is $\geq N - m - 1$ which is $\geq P(m)$. Thus the number of faulty attributes in the failure function is less than or equal to the number

of non-faulty attributes in the failure function. □

An obvious approach to the problem of finding the faulty and non-faulty attributes under compensation would be as follows. We start by identifying a non-faulty attribute. Based on this non-faulty attribute and the axioms in the failure function involving this non-faulty attribute we obtain additional non-faulty attributes. We continue in this manner until no more non-faulty attributes are obtained. Based on this set of non-faulty attributes and the axioms absent from the failure function we find some faulty attributes. When no more faulty attributes can be found we use this set of faulty attributes to obtain additional non-faulty attributes. We keep switching between the two sets until all faulty attributes have been obtained. Unfortunately, finding non-faulty attributes starting from faulty ones is difficult here, and the mechanism needed depends on the nature of compensations allowed. This gives rise to different algorithms for different types of compensations, which is a drawback.

We thus take a different approach. We first partition the set of attributes in $\Gamma(h(f))$ into two subsets, say X and Y, such that all the faulty attributes go to one subset and all the non-faulty ones to the other. Notice that during partitioning, we do not know whether the subset X or the subset Y contains faulty attributes. This classification is done in the second step. The advantage of this approach is that every attribute in the failure function is grouped with attributes possessing the same properties. This helps in the identification of non-faulty attributes in case of both restricted and unrestricted compensations in a uniform way.

Given an instance of a data structure D affected by fault f we first obtain the fault graph $G_D(f)$. The edges present in the fault graph correspond to the axioms that are valid. We then obtain all the connected components of $G_D(f)$ denoted $C_1, \ldots, C_n$. Notice that each connected component contains nodes corresponding to only faulty attributes or only non-faulty attributes. This is due to the fact that a valid axiom has either both attributes faulty or both attributes non-faulty.

To describe the properties exhibited by the connected components, we need one more

definition.

Definition 4.3.1 *Let $S = xy$ be an axiom relating attributes x and y. Then S is called a compensation detection (cdetection for short) axiom, with respect to a fault f, if $x, y \in \Gamma(h(f))$ but S itself is not present in $h(f)$.*

In the next two lemmas we provide a bound on the number of connected components containing non-faulty attributes.

Lemma 4.3.2 *Let C be a connected component containing vertices corresponding to non-faulty attributes. If w is a vertex corresponding to some non-faulty attribute present in the failure function and $w \notin C$, then for all $v_i \in C$ $dist(w, v_i) \geq \frac{m}{2} + 1$.*

Proof: Let the vertices in C be ordered with respect to the vertex numbering and let x, y denote vertices that are farthest apart (i.e. $dist(x, y)$ is maximum of the distances between all pairs of vertices present in C). If any vertex $v_i \in C$ has $dist(w, v_i) \leq \frac{m}{2}$ then either $dist(w, x) \leq \frac{m}{2}$ or $dist(w, y) \leq \frac{m}{2}$. Thus it is sufficient to consider only $dist(w, x)$ and $dist(w, y)$.

Now $dist(w, x) > \frac{m}{2}$ otherwise there would be an edge connecting vertices w and x (due to lemma 3.3.8) and $w \in C$ a contradiction. By a similar argument $dist(w, y) > \frac{m}{2}$. Thus for all $v_i \in C$, $dist(w, v_i) \geq \frac{m}{2} + 1$. □

Lemma 4.3.3 *There can be at most two connected components containing non-faulty attributes in the failure function of any fault, compensated or otherwise.*

Proof: By contradiction. Assume that there are more than two, say three, connected components containing non-faulty attributes. It follows from lemma 4.3.2 that there is no non-faulty attribute within a distance of $\frac{m}{2}$ from any vertex of such a connected component. Further, from lemma 3.3.8 each of the $\frac{m}{2}$ faulty attributes can be within a distance of $\frac{m}{2}$ from at most two connected components containing non-faulty attributes. Thus there must be at least $\frac{m}{2}$ faulty attributes between any two such connected components. Thus the total

number of faulty attributes is at least $3 \times \frac{m}{2}$. But this contradicts the fact that there are at most m faulty attributes. Thus there are at most two connected components containing non-faulty attributes. □

Corollary 4.3.1 *In a compensation free system there are at most two connected components.*

Proof: In the absence of compensations, Γ(h(f)) contains only non-faulty attributes and from lemma 4.3.3 there can be at most two connected components containing non-faulty attributes. □

We do not know the axioms that have been regenerated due to compensations. Thus we neither know the number of faulty attributes present in the failure function nor the manner in which they are distributed among the connected components. We only know the maximum number of faulty attributes ($P(m)$) that can be present in the failure function. However, lemma 4.3.3 is useful in characterizing the way in which *cdetection* axioms interconnect the connected components. Such a characterization is very useful in identifying the connected components containing faulty attributes.

It is clear that there can be more than one *cdetection* axiom involving attributes belonging to two different connected components. Since we are only interested in the existence of such a *cdetection* axiom rather than their actual number, the problem of identifying the connected components containing non-faulty attributes is simplified by transforming it into the following graph problem.

Let $G_C = (V_C, E_C)$ be a graph where a vertex v_i denotes the connected component C_i and an edge v_iv_j is present in the graph if there exists a *cdetection* axiom S such that $\Gamma(S) = \{x, y\}, x \in C_i,\ y \in C_j$. (We ignore those *cdetection* axioms that result in a self loop.) Further, we label an edge v_iv_j as

type 1 if C_i consists of faulty attributes and C_j consists of non-faulty attributes or vice versa.

type 2 if both C_i and C_j contain faulty attributes.

Type 1 edges can be used to partition (as will be shown later) the set of connected components into two distinct partitions such that one partition contains the connected components containing non-faulty attributes and the other contains connected components containing faulty attributes. To make this partitioning possible we eliminate all type 2 edges present in graph G_C. This is accomplished by algorithm Reduce and is discussed next. Algorithm Reduce identifies type 2 edges and merges nodes that are connected by a type 2 edge. The algorithm works in two stages. In the first stage it identifies certain nodes as candidates for merge. The identification is possible since we know the maximum number of faulty attributes that can be present in the failure function. The actual node merging and the construction of the reduced graph is done in the second stage.

Algorithm 4 *Reduce;*

Input: Graph G_C.
Output: A reduced graph $G_R = (V_R, E_R)$
begin
$V_1 \leftarrow \{\}$; (* Initialize sets V_1 and V_2 *)
$V_2 \leftarrow \{\}$;
(* set V_2 contains nodes that are candidates for merge *)
for each $v_i \in V_C$ **do**
begin
(* find the number of attributes in connected components corresponding to *)
(* nodes adjacent to the node under consideration *).
n$\leftarrow \sum_{v_k v_i \in E_C} |C_k|$;
(* if the number of attributes in neighbouring connected components is more*)
(* than $P(m)$ this connected component contains faulty attributes *).
(* as proved later in lemma 4.3.4 *)
(* Mark this node as a candidate for merge *)
if n $> P(m)$ **then** $V_2 \leftarrow V_2 \cup \{v_i\}$
else $V_1 \leftarrow V_1 \cup \{v_i\}$
end ;
$V_R \leftarrow V_1$;
$E_R \leftarrow E_C | V_1$;
(* E_R is the set of edges in E_C restricted to vertices in V_1 *)
if $V_2 \neq \phi$ **then**
(* check if there are any nodes to be merged *)
begin
$u \leftarrow \bigcup_{v_i \in V_2} v_i$;

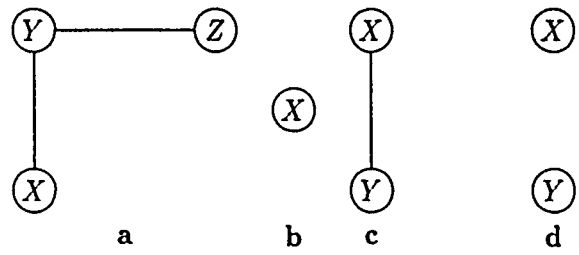

Figure 4.1: The four possible forms of reduced graph G_R

```
        (* obtain a new node by merging nodes in set V2. *)
        VR ← VR ∪ {u};
        (* add type 1 edges between the new node and each existing node *)
        (* of the reduced graph *)
        for each vi ∈ V1 do
            add uvi to ER;
        end ;
end .
```

The only edges in the reduced graph are type 1 edges (proved later in lemma 4.3.4). The graph G_R has one of the four possible forms shown in the figure 4.1.

We now partition the vertices belonging to V_R using type 1 edges so that one set contains vertices corresponding to connected components containing non-faulty attributes and the other contains vertices corresponding to connected components containing faulty attributes. If the graph is disconnected we merge the vertices into one set. We obtain a disconnected graph only when compensation does not take place. Thus the vertices correspond to non-faulty attributes and no partitioning is required. Otherwise, we pick a vertex with the maximum degree from the graph G_R and assign it to one of the sets, say set1. All vertices adjacent to this vertex are then put in the other set, say set2. We then obtain the attributes from the connected components corresponding to these vertices. The partitioning algorithm is given below.

Algorithm 5 *Partition;*

Input: Graph $G_R = (V_R, E_R)$.
Output: Two sets, one containing faulty attributes
and the other containing non-faulty attributes.

Only the attributes covered by $h(f)$ are covered by these sets.

begin
  if $E_R = \emptyset$ **then**
  begin
    set1 $\leftarrow \bigcup_{v_i \in V_R} v_i$;
    set2 $\leftarrow \{\emptyset\}$
  end
  else
  begin
    pick a vertex v_i such that v_i has maximum degree;
    (* assign the first vertex to one of the sets *)
    set1 $\leftarrow \{v_i\}$;
    (* Since reduced graph contains only type 1 edges, vertices connected *)
    (* by an edge should belong to different sets *)
    set2 $\leftarrow \{v_j | v_i v_j \in E_R\}$;
  end
  setx $\leftarrow \bigcup_{v_i \in set1} C_i$;
  (* C_i is connected component corresponding to vertex v_i *)
  sety $\leftarrow \bigcup_{v_i \in set2} C_i$;
end .

Notice that even after partitioning, in some cases, we still do not know which one of the two sets contains non-faulty attributes. We now present the complete algorithm for identifying faulty attributes. The algorithm requires as input a faulty instance of a data structure. It first obtains the connected components of the fault graph. It then constructs a graph using these connected components and cdetection axioms. The graph is then reduced and the vertices in the reduced graph partitioned into two sets. We then identify one of the two sets as the set containing non-faulty attributes. Lemma 4.1.2 provides a method of obtaining all the faulty attributes using the set of non-faulty attributes in the failure function.

Algorithm 6 *Identify;*

Input: a. Data structure specification.
  b. Instance of a data structure under fault f.
Output:
  faulty: the set of faulty attributes.
  nfaulty: the set of non-faulty attributes.
begin

```
{C1, C2, ...} ← connected components of G_D(f);
construct graph G_C = (V_C, E_C);
(V_R, E_R) ← reduce(V_C, E_C);
{setx, sety } ← partition(V_R, E_R);
(* Identify the set containing the non-faulty attributes present in the
failure function. *)
case
   (* Since P(m) is less than or equal to the number of non-faulty attributes in the *)
   (* failure function, a set containing more than P(m) attributes must contain *)
   (* non-faulty attributes *)
   |setx| > P(m) : nfaulty ← setx;
   |sety| > P(m) : nfaulty ← sety;
   (* Since there are at most P(m) faulty attributes in h(f), it follows
   from lemma 4.3.1 that a set having
   P(m) attributes contains non-faulty attributes if the other set has less
   than P(m) attributes. *)
   |setx| = P(m) and |sety| ≤ P(m) − 1 : nfaulty ← setx;
   |sety| = P(m) and |setx| ≤ P(m) − 1 : nfaulty ← sety;
   |setx| = P(m) and |sety| = P(m):
         begin
            (* find the non-faulty attribute x_i absent in h(f) *)
            (* x_i must be non-faulty since all faulty attributes *)
            (* are covered by either setx or sety *)
            x_i ← AT - (setx ∪ sety);
            (* find a neighbour of this attribute. *)
            pick S ∈ γ(x_i);
            x_j ← Γ(S) − x_i;
            (* neighbour belongs to set containing faulty attributes. *)
            if x_j ∈ setx then nfaulty ← sety ∪{x_i};
              else   nfaulty ← setx ∪{x_i};
         end ;
end  (* case *);
faulty ← {};
(* identify faulty attributes based on non-faulty attributes extracted
from failure function. *)
for each a in nfaulty do
   faulty ← faulty ∪{b|Γ(S) = (a, b), S ∈ (AX − h(f))};
end .
```

Before we provide the proof of correctness of the algorithm, let us first consider an example. Let us assume that we have a data structure with 16 attributes ($N = 16$) that is designed for 8-correctability ($m = 8$). Hence, the graph corresponding to the data structure has edges of distance up to 4. Since the total number of attributes is not greater than $2m$, the number of compensations should be restricted in order to obtain the faulty

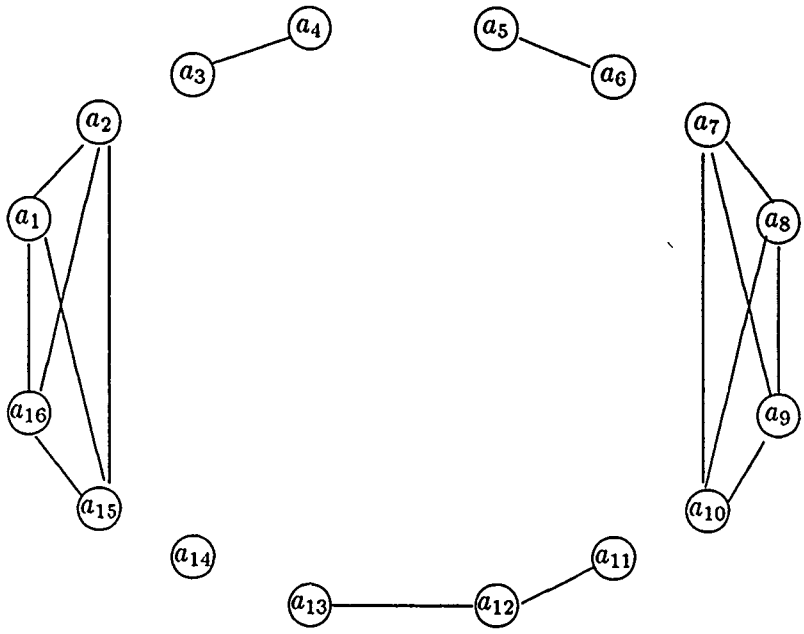

Figure 4.2: Fault graph of the example data structure

attributes. For the example, P(m) = $N - m - 1$ is 7. Consider a 8-ary fault which results in the fault graph shown in fig 4.2. We find the connected components of the fault graph. The 5 connected components are A =$\{a_1, a_2, a_{15}, a_{16}\}$, B = $\{a_3, a_4\}$, C = $\{a_5, a_6\}$, D = $\{a_7, a_8, a_9, a_{10}\}$ and E = $\{a_{11}, a_{12}, a_{13}\}$. The transformed graph is shown in figure 4.3. The vertices are labeled by component names. Applying the reduction algorithm to the graph in figure 4.3 we obtain the reduced graph shown in fig 4.4. Notice that vertices **a**, **b** and **d** in fig. 4.3 are candidates for merge since they are adjacent to vertices corresponding to components which together contain more than P(m) attributes. The edge connecting the two vertices **a** and **b** is a type 2 edge which is eliminated in the reduced graph. The vertex **a** in the reduced graph is obtained by merging the vertices **a**, **b** and **d** of figure 4.3. After partitioning, we obtain the two sets as setx = $\{a_1, a_2, a_7, a_8, a_9, a_{10}, a_{15}, a_{16}\}$ and sety = $\{a_3, a_4, a_5, a_6, a_{11}, a_{12}, a_{13}\}$. Now we identify setx as the set containing the non-faulty attributes since setx has 8 attributes and sety has 7 attributes. This corresponds to the third clause in the case statement in algorithm Identify.

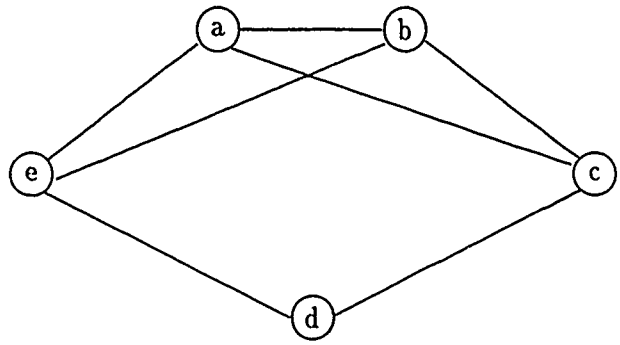

Figure 4.3: The transformed graph of the given fault graph

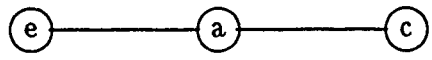

Figure 4.4: Reduced graph of the example

4.3.3 Correctness of the Algorithm

We now prove that the algorithm works correctly. We have to show that the attributes identified as faulty are indeed faulty and those identified as non-faulty are indeed non-faulty. Notice that we start with the attributes obtained from the failure function and extract the non-faulty attributes from this set. Then based on the non-faulty attributes extracted, we obtain the faulty attributes using lemma 4.1.2.

We start by obtaining the connected components from the fault graph. Since an axiom consisting of a faulty and a non-faulty attribute cannot appear in $h(f)$, the edge corresponding to such an axiom cannot be present in the fault graph. Thus every connected component must contain only faulty attributes or only non-faulty attributes. We need to show that faulty and non-faulty attributes do not get mixed during graph reduction and partitioning.

Lemma 4.3.4 *Algorithm Reduce eliminates every type 2 edge.*

Proof: From lemma 4.1.2 it follows, that there is an edge in graph G_C between a vertex corresponding to a connected component containing faulty attributes and a vertex corresponding to connected component containing non-faulty attributes. From lemma 4.3.3 there are at most two connected components containing non-faulty attributes. Let us consider each of the two possibilities (i.e., 1 or 2 connected components containing non-faulty attributes).

Case 1: Let a be the vertex corresponding to the only connected component containing non-faulty attributes. Since there are at most $P(m)$ faulty attributes in $h(f)$, it follows from lemma 4.3.1 that the number of attributes in this connected component is at least $P(m)$. The number of attributes in every other connected component is at least two since connected components are obtained from fault graph. Let there be a type 2 edge e joining two vertices representing connected components containing faulty attributes. Each vertex of e is adjacent to a (follows from lemma 4.1.2). The number of attributes in connected

components corresponding to vertices adjacent to each vertex of edge e is greater than $P(m)$. This is because each vertex of e is adjacent to a, which has at least $P(m)$ attributes, and to the other vertex of e which has at least two vertices. Thus both the vertices of edge e will be in set V_2.

Case 2: Let there be two connected components containing non-faulty attributes. Any attribute in one of these connected components is distance $\frac{m}{2}+1$ away from any attribute in the other connected component. Let the nodes corresponding to these connected components be a and b. Now all the faulty attributes lie within a distance of $\frac{m}{2}$ from each of the connected components containing non-faulty attributes. This is because there are at least m faulty attributes neighbouring a connected component containing non-faulty attributes and the total number of faulty attributes is m. Thus the vertex corresponding to a connected component containing faulty attributes will be adjacent to both a and b. Since there are at most $P(m)$ faulty attributes in $h(f)$, it follows from lemma 4.3.1 that the number of attributes in connected components corresponding to a and b is at least $P(m)$. Let there be an edge e connecting two vertices corresponding to connected components containing faulty attributes. Each vertex of e is adjacent to both a and b. The number of attributes in connected components corresponding to vertices adjacent to each vertex of edge e is greater than $P(m)$. This is because each vertex of e is adjacent to both a and b, which have at least $P(m)$ attributes, and to the other vertex of e which has at least two vertices. Thus both vertices of edge e are in V_2.

Thus every vertex which has a type 2 edge incident on it will be in V_2. Since the reduction process merges two vertices connected by a type 2 edge the proof is complete. □

Lemma 4.3.5 *The reduced graph is a bipartite graph if the failure function contains faulty attributes.*

Proof: There is no edge between two vertices corresponding to connected components containing non-faulty attributes. It follows from lemma 4.3.4 that the reduced graph contains no edge between two vertices corresponding to connected components containing faulty

attributes. Thus the only edges are between a vertex corresponding to a connected component containing non-faulty attributes and a vertex corresponding to a connected component containing faulty attributes. Hence the reduced graph is a bipartite graph. □

Lemma 4.3.6 *The partition algorithm divides the attributes intc two sets, one containing only faulty attributes and the other containing non-faulty attributes.*

Proof: Assume that the failure function does not contain any faulty attribute. In this case the reduced graph is disconnected (follows from lemma 4.3.3 and corollary 4.3.1). The partitioning algorithm merges all attributes present in the failure function in to one set and leaves the other set empty, which proves the lemma.

Assume that the failure function contains faulty attributes. From lemma 4.3.5 the reduced graph is a bipartite graph. From lemma 4.1.2 it follows, that the bipartite graph is connected. From lemma 4.3.4 it follows that the reduced graph contains only type 1 edges. Since partition algorithm puts vertices connected by an edge in G_R into separate sets the proof is complete. □

Lemma 4.3.7 *The algorithm identifies faulty attributes correctly.*

Proof: From lemma 4.3.6 the attributes present in the failure function are partitioned into two sets, one containing only faulty attributes and the other non-faulty attributes. The failure function contains at least $P(m)$ non-faulty attributes and at most $P(m)$ faulty attributes (follows from lemma 4.3.1). The attributes could be divided among the two sets in one of the three possible ways

1. At least one set has more than $P(m)$ attributes. If such a set exists then it is the set containing non-faulty attributes present in the failure function since there are at most $P(m)$ faulty attributes in the failure function.

2. At least one set has exactly $P(m)$ attributes and the other has less than $P(m)$ attributes. The set containing $P(m)$ attributes is the set of non-faulty attributes present

in the failure function. This is because the number of faulty attributes in the failure function is less than or equal to the number of non-faulty attributes in the failure function (follows from lemma 4.3.1).

3. Both the sets contain exactly $P(m)$ attributes. This can happen only if one non-faulty attribute is absent in the failure function. In this case the non-faulty attribute is adjacent to every faulty attribute.

Thus the algorithm extracts the set containing the non-faulty attributes present in the failure function correctly. From lemma 4.1.2 it follows, that all the attributes identified as faulty on the basis of set of non-faulty attributes present in the failure function are indeed faulty and the others non-faulty. This proves the lemma. □

Lemma 4.3.8 *The algorithm is optimal with respect to the number of axioms and identifies faulty attributes of a compensated fault in* $\mathbf{O}(mN)$ *steps.*

Proof: The connected components can be found in time linearly proportional to the number of edges in the graph. Since the number of edges in the fault graph is equal to the number of axioms, we require $\mathbf{O}(mN)$ time to find the connected components. Constructing the graph from the connected components requires scanning each axiom in the set $AX - h(f)$ once. This also can be done in $\mathbf{O}(mN)$ steps. The graph reduction works on graph G_C. Since the edges of G_C are obtained from axioms in $AX - h(f)$ and each edge is processed at most twice the reduction also takes $\mathbf{O}(mN)$ steps. Algorithm Partition takes constant time. The identification of a set as faulty or non-faulty takes constant time. Once the non-faulty attributes have been extracted from the failure function the identification of faulty attributes takes $\mathbf{O}(mN)$ steps (as proved earlier). Thus the entire algorithm executes in $\mathbf{O}(mN)$ steps.

The algorithm requires a constant amount of time to process every axiom. Since the data structure design does not use any redundant axiom, every axiom is significant. Thus the algorithm is optimal with respect to the number of axioms in the data structure. □

4.4 Correcting Faulty Attributes

In the above, we only considered the problem of identifying the faulty attributes. The next step is the correction, which is also based on the data structure axioms. Notice that since we have assumed that the axioms are complete, any correction that satisfies the axioms must be acceptable. The correction requires an interpretation of the axioms, since it is easy to see that with uninterpreted axioms, the problem of determining attribute values that would satisfy the axioms is undecidable. The interpretation is usually easy to provide. For example, consider an axiom in which at least one (of the two) attributes is generic. We could then cast this axiom in form of a "correction procedure" that walks through the data structure using the generic attribute and computes the value(s) of the other attribute. This is possible because of the restrictions we imposed on the axioms. We now give the correction details.

We start by identifying all the faulty attributes. From lemma 4.1.2 it follows that every faulty attribute has a non-faulty attribute as a neighbour in its failure function. So let x be the faulty attribute, and y one of its neighbouring nonfaulty attributes connected via an axiom, say $A(x, y)$. If we correct the faulty attributes sequentially, it is easy to show that one can always choose an order such that the nonfaulty neighbour (which we denoted as y) is a generic attribute. Thus we can use the interpretation of the axiom $A(x, y)$ and correct the attribute x. The type of the attribute x does not matter because of the required restrictions on the axioms. If fact, even if x represents an atomic attribute that has been collapsed with a weak generic attribute, both of them can be corrected independently using the attribute y. Thus we need not distinguish between a fault in a weak generic attribute and its associated atomic attribute, as claimed earlier.

Chapter 5

Applications

In this chapter we study three applications of the theory developed so far. In the first application we apply our design methodology to local correction. In the second application we extend our method to data integrity and in the third application we study the design of a robust file system.

5.1 Global and Local Correction

In the example in section 3.3.1.2 we chose an atomic attribute (the *count*) that is global in nature. To validate an axiom based on this attribute we have to go through the entire instance of the data structure. The time taken is O(n) where n is the number of elements in the list. Hence testing the integrity of the data structure for every operation is prohibitive in time. Not testing the integrity of the structure every time an operation is performed, may also lead to anomalous situations. Consider an instance of the linked list as shown in figure 5.1.

In the structure shown in figure 5.1, a's forward pointer is incorrect and points to c. Now if a new element a' is inserted after a but before c then the resulting structure would be as shown in figure 5.2.

Notice how b has become inaccessible using either of the pointers, even though there was only one error in the original structure. Assuming that no more errors occur, this error

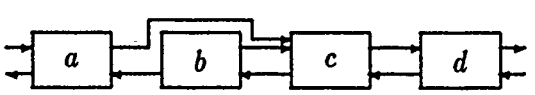

Figure 5.1: Data structure with one invalid pointer.

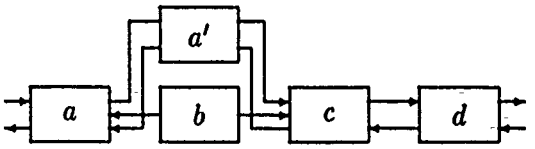

Figure 5.2: Figure showing inaccessibility of b.

would be detected in one of periodic checks of the integrity of the structure, however, the error would show up (improperly) in the count field that would then be adjusted. But the element b is lost forever.

This suggests that instead of periodic checking, the structure should be checked every time an operation is performed. Since checking the entire structure takes too much time, local checking should be employed. We define **neighbourhood** of an element in a data structure as that subsection of the data structure whose attribute values have to be changed to accommodate a new element or to delete an existing element. If the elements in the neighbourhood of the place of update are correct then an update (insert or delete) in this neighbourhood would not affect the number of errors present (if any) in the structure. If there is an error in the neighbourhood of update then the correction is applied before the update. This way the structure's integrity is always maintained and at any time it is possible to obtain the correct structure from an erroneous one. We explain this with an example on doubly linked lists where attributes are introduced gradually as and when required.

5.1.1 Local Correction in Linear Data Structures

We start the design of a locally correctable list **F** with the initial structure of section 3.3.1.2 consisting of $AT = \{fp, bp\}, S = \{S_1\}$ and $F = \{f_1, f_2\}$. However, we now add an additional pointer to achieve 1-correctability. We shall refer to this pointer as an alternate pointer (ap). The two additional axioms are

$$S_2 : \forall E \in EL_P[ap(E) = fp^2(E)] \tag{5.1}$$

attribute	S_1	S_2	S_3	S_4	S_5	S_6
fp	X	X		X		
bp	X		X		X	
ap		X	X			X
rp				X	X	X
fp, ap	X	X	X	X		X
fp, rp	X	X		X	X	X
bp, ap	X	X	X		X	X
bp, rp	X		X	X	X	X
ap, rp		X	X	X	X	X

Table 5.1: Failure function for locally correctable doubly linked list

$$S_3 : \forall E \in EL_P \exists E' \in EL_P : [ap(E) = E' \Leftrightarrow bp^2(E') = E] \tag{5.2}$$

It is easily seen that these axioms distinguish all single faults. Also, since every failure function has at least one axiom involving a generic attribute, correction is feasible. We can make the above data structure 2-correctable by adding yet another pointer that we shall call the reverse pointer (rp). Then, $AT = \{fp, bp, ap, rp\}$, and $AX = \{S_1, \ldots, S_6\}$, as given below:

$$S_1 : \forall E \in EL_P [fp(bp(E)) = E \wedge bp(fp(E)) = E] \tag{5.3}$$

$$S_2 : \forall E \in EL_P [ap(E) = fp^2(E)] \tag{5.4}$$

$$S_3 : \forall E \in EL_P \exists E' \in EL_P : [ap(E) = E' \Leftrightarrow bp^2(E') = E] \tag{5.5}$$

$$S_4 : \forall E \in EL_P \exists E' \in EL_P : [rp(E) = E' \Leftrightarrow fp^2(E') = E] \tag{5.6}$$

$$S_5 : \forall E \in EL_P [rp(E) = bp^2(E)] \tag{5.7}$$

$$S_6 : \forall E \in EL_P [rp(ap(E)) = E \wedge ap(rp(E)) = E] \tag{5.8}$$

Table 5.1 gives the failure function for the locally correctable doubly linked list (X indicates that the axiom is violated).

The set of axioms is sound, complete and proper. However, since rp and ap have a span of two, we need two headers. The example also highlights the advantages of local

correction. Local correction overcomes the problems of internal compensation. Notice that we can tolerate up to two faults in the neighbourhood of a node. However, more than two faults can also be corrected provided the number of faults in any neighbourhood does not exceed two. Moreover, the corrections in the two neighbourhoods can be done concurrently. The time taken to perform correction is constant. For the example given above the size of the neighbourhood is 5. Thus correction involves updating the attributes in at most 5 elements. In the case of global correction we need to traverse the entire data structure to perform correction.

5.1.2 Local Correction in Nonlinear Data Structures

Local correction can be applied to other structures like Trees. Taylor et al [TMB80] have used chain and thread pointers in those nodes where the regular tree pointer is not present. Their method is space efficient as they do not use any additional storage but requires guessing to perform correction [TB85]. Davis [Dav87] provides a method for AVL trees. They use local correction and their method does not degrade the performance of other operations on AVL trees. Sampaio et. al. [SS85] suggest the use of a parent pointer and a postorder pointer as the alternate path to each node. In this method, correction takes a lot of time and also uses additional storage space for pointers. We also adopt a method that requires additional space for the pointers but correction becomes simple and localized. We connect each parent to all its children by a pointer called the **family pointer**. This forms a list with the parent as the header. A tree with family pointers is shown in figure 5.3. (Neighbourhood of a node consists of the node along with its children).

We shall analyze the structure by our method. The attributes are $AT = \{tp[0..N-1], fp\}$ where $tp[0..N-1]$ are the tree pointers to the N children. We assume that if ith child does not exist, $tp[i]$ simply points to itself. For notational convenience, we also assume that $tp[i]$ points to itself for $i < 0$ and $i > N-1$. We also have a *family pointer*, denoted fp that goes through the parent and all its children. The only axiom is:

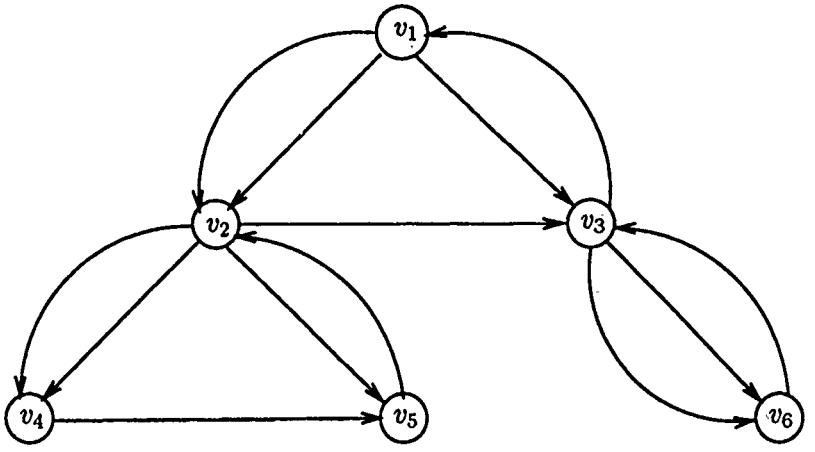

Figure 5.3: Locally correctable tree.

$$S_1 : \forall E_1, E_2 \forall i [tp[i](E_1) = E_2 \Leftrightarrow fp(tp[i-1](E_1)) = E_2] \tag{5.9}$$

S_1 relates the parent node to the nodes corresponding to the ith and $(i-1)$st child as follows. If the ith child of element E_1 points to E_2 then the pointer fp in the element corresponding to the $(i-1)$st child of E_1 points to E_2. The possible faults are $F = \{f_1, f_2\}$ and the failure function $h(f_1) = h(f_2) = \emptyset$ where

f_1 = One or more tree pointers pointing incorrectly.
f_2 = family pointer pointing incorrectly.

Obviously, the two faults are not distinguishable. So we convert the chain of family pointers into a circular one and add another attribute count at each node that denotes the number of children this node has. We denote this attribute as $count(E)$ for element E. Thus $AT = \{tp, fp, count\}$

$$S_2 : \forall E[count(E) = \sum_{i=0}^{N-1} ord(tp[i](E) \neq E)] \tag{5.10}$$

$$S_3 : \forall E[fp^k(E) = E, k = count(E)] \tag{5.11}$$

S_2 states that the number of children for any node is equal to the number of pointers that do not point to itself. S_3 states that starting from any element we can return to it by following k instances of the attribute fp, where k is equal to *count*. In order to make the axiom set complete, we need to add a conjunct similar to C_1 of section 3.3.1.2 to the term relating fp and *count*. Again, since we relate an atomic attribute *count* with generic attributes fp and tp we face the problem of internal compensation. To overcome this problem we need an additional weak generic attribute and an additional term similar to C_1' of section 3.3.1.2. We ignore these in the discussion below with the understanding that C_1 is maintained implicitly and C_1' is added explicitly. The set F is also augmented with fault f_3 denoting improper count field. The failure functions are given below.

$$h(f_1) = \{S_3\}$$
$$h(f_2) = \{S_2\}$$
$$h(f_3) = \{S_1\}$$

Consider a locally correctable B-tree as in [TB86]. We can generate axioms similar to the previous example to obtain a robust B-tree structure for a desired level of correctability. Let each node in the tree represent another data structure, say a list, with the node as the header. We consider the list to be at the next level of nesting of the compound data structure. We can now make the list robust to the desired level of correctability. The axioms for the list are totally independent of the axioms generated for the B-tree structure. Notice that the list and the B-tree are both locally correctable. Thus our method caters also to **hierarchical data structures.** In fact we can generalize our method to a data structure of any desired depth of nesting. Such a structure allows concurrent repairs for nodes at level i provided the integrity of the structure at level $i-1$ is established. It is necessary to establish the integrity of the structure at level $i-1$ since we do not have external pointers to the distinguished elements (for e.g., headers for a list) of the data structure at level i. In other words level i elements can be accessed only by the access path provided at level $i-1$. We do not require the structure at any level to be homogeneous. Thus one node of a B-tree could represent a tree, while another could represent a list and so on.

One problem not solved by the local correction scheme described above concerns the search that typically precedes an insert or delete operation for locating the appropriate place of insertion/deletion in the structure. In general, it is necessary to do local detection as the search procedure *walks through* the structure. The complexity of detection in this phase is governed by the number of elements examined by the search procedure (e.g., O(log$n$) for a balanced tree). This also means that the neighbourhood at the beginning of the search point (e.g. the root of the tree) will be checked most frequently and the frequency of checks will decrease as we go further down. It is possible to take advantage of this observation to reduce integrity checking overhead.

5.2 Data Integrity

We can view data integrity in the same manner as structural integrity. However, in case of data, the semantics attached to the values is not known. Also, the data by itself may not have any relationship with the data elements stored in other parts of the structure. We can have redundant information by using additional attributes for generating relationships among data elements. Note that even though data is organized on the basis of the structural attribute the data axioms will not use this structural attribute.

Since we need to create attributes to relate groups of data items, the first problem is to organize data such that we need minimum additional space for a given level of correctability. We shall assume that the data is to be stored in a parallel piped like structure. We also assume that the data elements could be related by some function. The function should be such that any single change in one of the arguments changes the value of the function. One example of such a function is the addition operator. We assume that the application of this function does not result in arithmetic exceptions. Under these constraints, we can show that the hypercube arrangement is optimum.

It should be noted that the underlying data structure that holds the data items of interest could be arbitrary (e.g., a list, tree, etc.), and its integrity can be ensured by using the techniques of previous sections. The parallel piped structure required for data integrity

could be implemented in two ways: (a) by using additional pointers, (b) storing elments in a N-dimensional array. In case (a) we must ensure structural integrity of the parallel piped structure as well. It is clearly possible to reduce the number of pointers used by merging the underlying structure and the parallel piped structure; however, for simplicity, we will not consider this refinement. In case (b), the underlying linked structure is constructed by using addresses of the array elements (instead of real pointers). Even though (b) is less flexible, it is much simpler, requires much less storage for pointers, and eliminates problems of general linked structures.

The next two lemmas characterize the number of data errors that can be detected and corrected if the data is arranged in a specific pattern, viz. as a hypercube.

Lemma 5.2.1 *Arranging the elements in a N-dimensional hypercube gives N detectability (but not $N+1$-detectability) and requires no greater storage than any other storage pattern that gives the same detectability.*

Proof: For the first part, notice that every element is a member of at least N dimensions. Hence every element is an argument to the function along each of the N dimensions. To change from one correct instance to another requires a change in one of the elements along with a change in each of the N functions that this element is an argument of, giving a total of $N+1$ changes and a detectability of N.

Now to see that it is not possible to obtain $N+1$-detectability, consider an instance with elements $a_1, \ldots, a_N$ along the N dimensions and some other element a_0 on one of the N dimensions. Let the errors in elements $a_0, a_1, \ldots, a_N$ be such that compensation takes place. As a result of compensation the function (say f) computing the value along each of the N dimensions is not affected. Also notice that no amount of duplication of attributes would help. This is because the function is a many to one function. It is also not possible to use a function (say f') that takes elements from different dimensions as arguments. This is because this function (f') behaves just like a function (f) with arguments from one dimension but with the axis rotated. Also it is not possible to relate the functions f and f'

for the same instance of data structure.

Next we show that no other structure can yield better detectability without using more storage. For this, we only need to invoke the fact that for a given volume and number of vertices, a regular structure has the least surface area. Let the total number of elements be Q. The number of elements (k) in one dimension if stored as a cube is $Q^{1/N}$. Let the elements be stored as some other structure in that some dimension has less than k (say i') elements. Thus to keep the the total number of elements constant the number of elements in some other dimension has to change to say j'. Now we require $i' * j' =$ constant and $i' + j'$ as small as possible (since this is the number of stored sums). Clearly the minimum is attained when $i' = j'$. Thus the cube structure requires no more space than some other structure. This completes the proof of the lemma. □

Lemma 5.2.2 *The N-dimensional cubic structure gives $N-1$ correctability (but no more).*

Proof: We shall prove the result by induction on N. For one correctability we require a 2-dimensional plane. We can store the sum of the elements along two dimensions (rows and columns). With this arrangement any single fault can be uniquely detected and corrected. This proves the basis step. Now assume the validity of the result for $N-1$ and consider a N-dimensional structure. We form a N-dimensional structure from $N-1$-dimensional structures so that every element is a member of at least N $N-1$-dimensional structures. Every element contributes to each of the N set of axioms, that are generated for every $N-1$-dimensional structure, of which it is a member. If there are at most $N-1$ errors (compensated or otherwise) then at least one of these substructures will not contain *all* the faulty elements. In other words, such a substructure will have at most $N-2$ errors. From the induction hypothesis these can be corrected. This proves the lemma. □

We shall give an example using a two dimensional structure. In the case of a 2-D structure we store the sum along two planes. The attributes are $AT = \{rsum_i, csum_j\}$ where $rsum_i$ denotes the sum of the elements in the i-th row and $csum_j$ denotes the sum of the elements in the j-th column. The axioms are:

$$S_1 : \forall i \sum_j data_{ij} = rsum_i \tag{5.12}$$

$$S_2 : \forall j \sum_i data_{ij} = csum_j \tag{5.13}$$

$$S_3 : \sum_i rsum_i = \sum_j csum_j \tag{5.14}$$

The failures are erroneous data/sum items. The failure functions are:

$$h(data_{ij}) = \{S_3\}$$
$$h(rsum_i) = \{S_2\}$$
$$h(csum_i) = \{S_1\}$$

Each element contributes to the *csum* of the column in which it lies. Similarly it contributes to the *rsum* of the row in which it lies. If there are two faulty elements and they do not lie in either the same column or same row then our axiom structure allows correction to the two faulty elements to be done concurrently and independently. This is similar to local correction discussed earlier.

5.3 Design of Robust File System

File system is the most vulnerable and critical component in any computer system. File systems are vulnerable because they operate in a somewhat imperfect environment. The immediate cause of failure, in file systems of contemporary design, is usually the loss of mapping information responsible for the control of the file system. As discussed by McGregor and Malone [MM81] the mapping information can be lost in one of the three ways:

1. The File Directory is damaged and cannot be read.

2. The free space map is corrupted and duplicate allocation of space occurs.

3. The file linkage information is lost.

Thus, even if the data by itself is not damaged, it may become inaccessible. We study the design of a file system that enhances the robustness of the mapping information. Any corruption of a data block results in the loss that data (i. e. the data is not recoverable). We assume that the file system is implemented on a disk based system and faults result in the loss of one or more sectors of information. We assume the sector size to be the same as block size (a block is the logical unit of information) and use these two terms interchangeably in the rest of the discussion. A fault is detected whenever reading a block results in a disk error. It is sufficient to consider only read operation for fault detection since the write operation is normally implemented as write in to the disk followed by a read of the same block. We assume that the detection is performed by the underlying hardware. We present the organization of the physical layer of the file system that tolerates a maximum of one fault in each of the three entities providing the mapping information.

5.3.1 Robustness of the File Directory

The File Directory is the single most important information in any file system. It usually contains information pertaining to the files present in the system, volume information and other accounting information. The simplest and the most efficient way to increase the robustness of the file directory is to duplicate the file directory. We employ duplication in a controlled fashion to be described shortly. We assume that the maximum number of files that can be stored on one volume is predetermined (not an unreasonable assumption, in fact most contemporary designs put such a limitation). We use a hash function to preallocate a block for each file that can be present in the system. This association is performed during volume initialization. The hash function uses the directory entry number as its argument and provides an address (i. e. the block number), where the duplicate directory associated with this file is to be stored. After volume initialization each such block contains the directory entry number and an indication that it is unused. The advantage of using the hash function is that the information regarding the location of the duplicates need not be stored. If sequential access of the directory information via the duplicate directory is

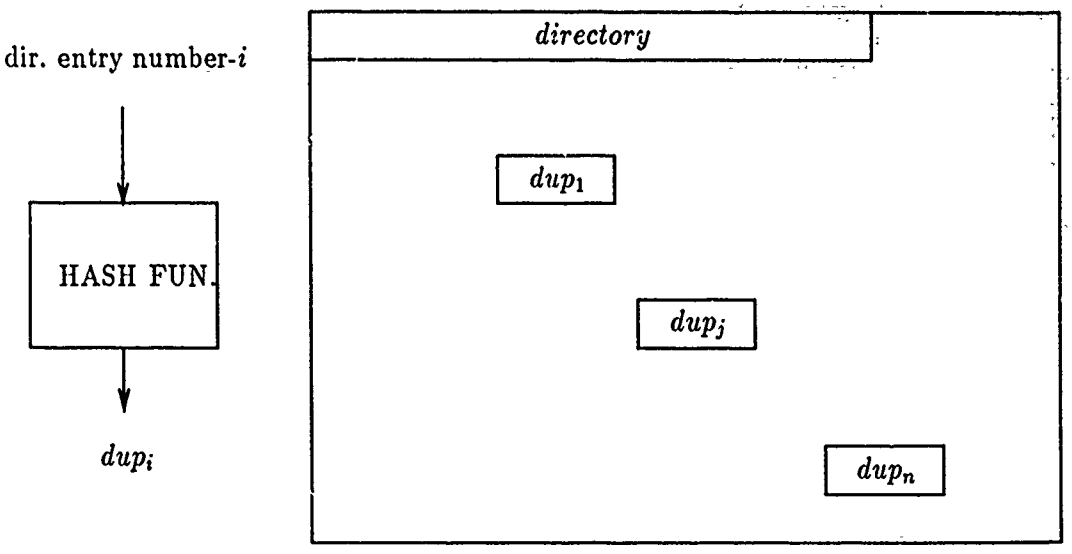

Figure 5.4: Directory for robust file system

desired, the blocks containing the duplicate directories could be linked together. Further, the linking of the duplicate directory blocks can be performed during volume initialization. Every file creation and deletion results in the update of both the directory and the duplicate directory block associated with that particular file.

It is not necessary to copy all the information from the main directory into the duplicate. For reasons of efficiency, we feel that it is sufficient to maintain the file name, the main directory entry number and linkage pointer information. With this arrangement, the duplicate needs to be updated only at file creation and file deletion. During other times, it is sufficient to update the linkage blocks and the information in the main directory.

If one of the duplicates is corrupted, it can be regenerated using the information in the main directory. Similarly, if the main directory is corrupted then by using the hash function to traverse through the duplicates, all the relevant information in the main directory can be recreated. The only information lost is the accounting information. The pictorial representation of a robust directory is shown in figure 5.4.

5.3.2 Robustness of the Free Space Map

The free space map is normally stored in the volume at a predetermined location and in the discussion we shall assume the same. To improve the robustness we do the following. In each block of the volume, space is reserved for a preamble. During volume initialization, all blocks are marked free. When a block is allocated the preamble is updated so that it contains pointers to the block containing duplicate directory information associated with the file to which this allocated block belongs and to the main directory. Every time a new block is allocated on the basis of the information in the free space map, the block is read and the preamble is checked to make sure the block is free. By our assumption, there can be at most one fault in the free space mapping information and this fault is detected by a read operation. If no error is encountered during a read of the newly allocated block we have two options:

- We check to see if the information in the preamble is correct, i. e. whether the entry number in the main directory and the block number of the duplicate match. If the information is incorrect we assume that the block was free (since we assume single fault in each of the three mapping information), i. e. the free space map information is correct. Otherwise we do what is discussed in the next option.

- We err on the side of safety and assume that the free space map is incorrect and update it accordingly. We also make note of the fact that the free space information was corrected. When the number of corrections of the free space map reaches a threshold we recreate the whole free space map and the block preamble on the basis of the information available in the file directory and linkage information.

The method discussed is inefficient and should be used only when reliability is of utmost importance.

5.3.3 Robustness of the Linkage Information

The file linkage information specifies the location of the data in the file. Since the file linkage itself is stored in the file system, it is also subject to the same faults as the data. Since file linkage information is essentially a set of pointers to data, loss of a block of file information results in a substantial loss of data. To enhance the robustness we use another set of pointers to data. We can view the file as a linked list with two pointers, where the data part of the linked list corresponds to the data blocks of the file and the two pointers are the two sets of linkage information. It follows from our knowledge of linked lists that robustness is best achieved by using the two set of linkage information in a manner analogous to the forward pointer and the backward pointer in a doubly linked list. Notice that the two sets of linkage information are stored separately. Separation of linkage information provides better robustness since a fault in one block can destroy only one piece of information (or one attribute as viewed from our model). Our example of linked lists in 3.3.1.2 requires that we have one additional attribute (or linkage information) to correct a corrupted block containing linkage information. The additional information is stored along with the data and contains the block number of the next data block. Let us call this as data linkage pointer (*dlp* for short). The data linkage pointer of the last block contains the entry number of the file in the main directory. Let us call the two sets of linkage pointers as forward linkage pointer (*flp* for short) and backward linkage pointer (*blp* for short). We now discuss the *flp* and *blp* in some detail.

To locate a byte of data we first obtain the block position, b, in which the data resides. We view both *flp* and *blp* to be an array. The jth entry of the *flp* array contains the address (block number) of the jth block. The block number of the jth block in the *blp* array can be found in the $(j+1)$st position. The last entry in the *flp* and the first entry in the *blp* points to the main directory. The axioms for the linkage information are as follows:

$$\forall 1 < i < max.blk.no.\,[flp(i) = blp(i+1)] \tag{5.15}$$

$$\forall 1 \leq i \leq max.blk.no.\,[dlp[flp(i)] = flp(i+1)] \tag{5.16}$$

$$\forall 1 < i < max.blk.no.\,[dlp[blp(i)] = blp(i+1)] \tag{5.17}$$

If the files are large, the amount of linkage information is substantial and may span several blocks. The linkage blocks themselves can be viewed as another file, or data structure, containing data block addresses. We view the linkage blocks themselves as a data structure and assume it is organized as a tree. In such a case, the higher level of linkage pointer points to a set of blocks each of which contains linkage information. In most cases, depending on block size, a tree depth of 3 is adequate. Organizing the linkage blocks as a tree structure results in faster access time for the data. However, a loss of one linkage block at a higher level of the tree translates into a loss of a very large amount of data. Thus we need to enhance the robustness of the tree organization. In a tree organization, the leaf level blocks correspond to either the *flp* or *blp* array discussed earlier. For the blocks in the non-leaf levels, the linkage information is data for that level and the *dlp* in this case corresponds to the family pointer of the example discussed in section 5.1.2 and the linkage information corresponds to the tree pointer of the same example. We can enhance the robustness of these non-leaf level linkage blocks by adding attribute *count* to each such block and making the *dlp* circular. We have one tree for *flp* and another for *blp* for each file.

Notice that this is a hierarchical data structure with robustness criterion and attributes for the non-leaf nodes that is different form those of the leaf nodes. It is also clear that both the data structures (at the leaf level and non-leaf level) are capable of local correction. The file linkage information can be depicted pictorially as shown in figure 5.5.

Let us discuss the space and access time overheads associated with our design. We confine our attention to access time only since computational overhead is negligible. Let us consider the space overhead first.

For each file we require one additional block to store the duplicate directory. If we employ redundancy for the free space map, we require a few bytes (space to accommodate one disk block number and directory entry number) to store the pointers. To enhance the robustness of file linkage information, in each data block we require space for storing the *dlp* (space to

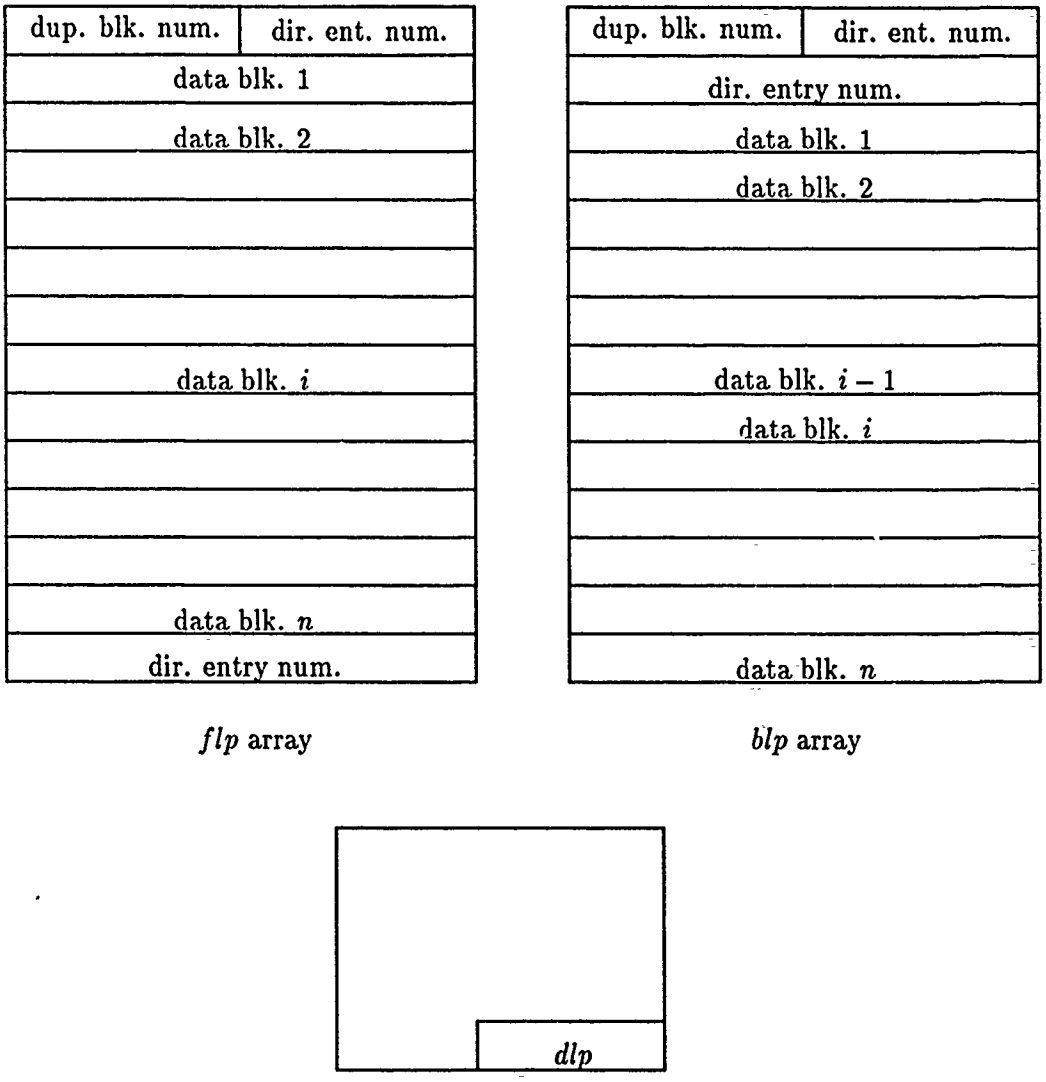

Figure 5.5: File linkage information

accommodate one disk block number). We also require space for the *blp*. We require enough space to accommodate one disk block number for each entry of *flp* and *blp* and we have one entry per data block. It is clear that the major factors contributing to space overheads are the space used for the linkage pointers and free space mapping. The additional linkage pointers require twice the space used by the linkage blocks in a non-redundant organization. The additional pointers employed for redundancy of free space mapping also require twice the space used by linkage blocks in a non-redundant organization. The linkage pointers are a small fraction (logarithmic amount) of the total space used by the file, since every block of data requires space for one entry of linkage information in a non-redundant organization. In a file organization which employs redundancy of linkage information only, the space used for linkage information per data block is three times the space used in a non-redundant organization. In a file organization which employs redundancy of linkage information and free space mapping, the space used for linkage information per data block is five times the space used for linkage information in a non-redundant organization.

Let us consider access times. The duplicate directory block need be written once during file creation and file deletion. Hence these operations require an additional write. An update of either *flp* or *blp* would entail a change in the other and would require two additional accesses (one for read and one for write). Use of *dlp* may require two additional access (one for read and one for write) if data movement in one block requires updating the *dlp* stored in another block. An assessment of access time overhead requires additional information specifying file access modes and their frequencies. It is our view that writes due to change in *dlp* will be responsible for most of the additional accesses. One way to overcome this drawback is to avoid using *dlp* and instead replicate either *flp* or *bip*. This change requires additional space (space required for non leaf nodes of the tree structure).

Use of *dlp* also complicates crash recovery procedures since it involves multiple writes. Complications arise since we need to update two blocks of information and the information content for the two blocks is different. One method of overcoming this difficulty is to use the following method. We write one of the blocks (the new data) into a free area of the

disk. This block number is the new value of *dlp*. We then write the new value of *dlp* and the modified linkage information that includes the new written block as part of the file in to the stable storage. The linkage blocks and the other block (whose *dlp* needs to be updated) can be changed using the information in the stable storage. An atomic update of either *flp* or *blp* requires a two phase write, i. e. , logging information on to a stable storage and then updating from the stable storage. Similarly, atomic update of directory, resulting from create and delete operations also requires a two phase write.

The time to identify the faulty attribute in the data structure (generated by the file linkage information) is **O**(1) if the failure functions are given. However, the time taken to evaluate the failure function itself is different for the different axioms. This is due to the fact that the data and the attributes are not stored together. Only *dlp* is stored along with the data. Thus evaluation of an axiom involving *dlp* requires reading all the data blocks, whereas evaluating an axiom involving *flp* and *blp* requires reading only the linkage blocks (the number of linkage blocks required is logarithmic with respect to the number of data blocks). The same is true for the correction algorithms also.

The fault model assumed in the design is simplistic. A better fault model would be one that mirrors the characteristics of contemporary disks. Some of the parameters of such a fault model include the failure rate of the sectors, probabilities of multiple sector failures in the same track, probabilities of multiple sector failure in the same cylinder, disk performance times etc. These characteristics should be used to design the placement of the redundant information discussed in our design. The placement of the redundant information affects the efficiencies of the different file operations. A more comprehensive design using these issues is in progress.

Chapter 6

Conclusions

In this thesis we have developed a formal and unifying method for looking at data and structural integrity. It was shown how 1-correctable structures can be synthesized systematically, starting with an initial specification of the data structure. We have also stated several results concerning the number of attributes and axioms needed and how the axioms should be generated.

Our model is very similar to the fault model used in [TMB80]. One difference is that we do not consider replacement faults as they have done. We require detectability of less than $2m$, to achieve m-correctability whereas their model requires detectability to be twice the desired amount of correctability. Both the methods require $m + 1$ generic attributes for m-correctability (they refer to it as $m + 1$ edge distinct paths). However, the main distinction between the two methods is in the manner that faults are considered. They develop their results using an Instance Fault Model, whereas we use the Generic Fault Model. As a result we are able to tolerate any number of faulty instances of the same attribute. This is particularly appealing when the software routine that updates a pointer does not work as required. In such a situation it is highly probable that more than one instance of an attribute is erroneous. This situation is very easily tolerated (and corrected) in our approach.

In the design approach discussed, axioms with more than two attributes are handled by imposing a structure on the axioms. However, as discussed in section 3.4 it is not necessary to impose any such restriction. We obtained results that generalize lemmas 3.3.8 and 3.3.9 for a system where each axiom has exactly k attributes and the required level of detectability m is even. However, these results have very limited practical use because of the restriction which requires exactly k attributes per each axiom.

We also presented an algorithm for identifying faulty attributes. The algorithm can be used on any data structure that is designed using the method given in lemma 3.3.8. The fault identification algorithm does not assume the existence of any precomputed fault syndrome table. Any mechanism that uses a fault syndrome table suffers from the drawback of requiring additional space to store this table. The size of the table is substantial. For a compensation free system we need a table with ${}^{N}C_{m}$ entries. The number of entries increases rapidly for a system with compensations. Our mechanism does not require any additional storage.

The number of computational steps required by our algorithm is proportional to the number of axioms in the data structure specification. Our algorithm is optimal with respect to the number of axioms in the data structure specification. This is because our algorithm uses every axiom and there is no redundant axiom in the data structure specification. We are unaware of any work which deals with compensated faults and are unable to provide any comparison with our algorithm.

The local correction, introduced in chapter 5, not only speeds up correction but also allows it to proceed concurrently in disjoint neighbourhoods. It is thus very useful in database applications. Lehman and Yao [LY81] have used a structure called B^*-link tree to allow concurrent operations on B-trees. This is achieved by using an additional pointer called **link pointer**. Since a link pointer is compatible with the *family pointer* introduced for local correction, we can not only increase the concurrency but also allow repair concurrently with normal operations.

As shown in chapter 5, the approach for ensuring structural integrity can also be applied to achieve data integrity. Unfortunately, it is not possible to consider the latter completely independent of the former since the specific arrangements of original and added data attributes (e.g., in form of a parallel piped) introduce the problem of ensuring structural integrity of the arrangement. For simplicity, we assumed that the integrity of this arrangement and that of original structure are dealt with separately.

6.1 Further Work

The design we have presented is biased towards correctable data structures or detectable data structures. It would be nice to explore designs which cater to both detectability and correctability. Also interesting to look for are mechanisms to integrate data integrity and structural integrity. It is not clear how to exploit the semantic information associated with data for purposes of fault tolerance.

We have been able to generalize lemmas 3.3.8 and 3.3.9 to a data structure with more than two attributes per axiom when the required level of correctability is even. It remains to be seen whether these lemmas apply even when the desired level of detectability is odd. We also need a method to synthesize a data structure with more than two attributes per axiom when the required level of correctability is odd. Another aspect to explore is data structures with axioms made of different number of attributes.

The ideas developed in this thesis could be used to enhance algorithmic robustness of the software as well. The usual mechanism for achieving algorithmic robustness involves multiversion software, [AC77] i.e., different program versions developed from the same specifications. The success of this approach rests on the degree of independence between the various versions. If the given abstract structure is implemented as two or more concrete data structures then it would be possible to write fairly different programs for the same underlying task. Moreover, in case of a structural fault in one of the concrete data structures (either caused by a bug in the software or by a transient hardware fault), the abstract structure will still be accessible via the other(s) and could be used for error detection/correction.

As an example, consider an application program working on a list of numbers. The abstract concept of list can be implemented either using an array or a linked list. The two implementations lead to substantially different data manipulation algorithms, thus providing independence. Moreover, if the pointers in the list-structure get corrupted, the array implementation could be used for detection and correction.

Some parts of the file design that was discussed is being implemented as a masters paper.

It would be interesting to look to the design of a file system kernel which allows users to tailor the reliability of their files. Robust files are becoming increasingly important with the advent of object oriented data bases which need to store persistent data.

It would be interesting to see if the model developed here can be applied outside the realm of data structures. Fault tolerant systolic arrays seems to hold some promise since systolic arrays have a regular structure like data structures.

Bibliography

[AC77] A. Avizienis and L. Chen. On the implementation of N-version programming for software fault tolerance during program execution. *Proceedings COMPSAC 77*, pages 149–155, Nov. 1977.

[AL81] T. Anderson and P. A. Lee. *Fault Tolerance Principles and Practice.* Prentice Hall, 1981.

[And85] T. Anderson. An evaluation of software fault tolerance in a practical system. In *Digest of Papers, FTCS-15*, pages 140–147, June 1985.

[Avi78] A. Avizienis. Fault tolerance: the survival attitude of digital systems. *Proceedings of IEEE*, 66:1109–1125, Oct. 1978.

[CA78] L. Chen and A. Avizienis. N-version programming: A fault tolerant approach to reliability of software operation. In *Digest of Papers, FTCS-8*, June 1978.

[CB81] F. Cristian and E. Best. Systematic detection of exception occurrences. *Science of Computer Programming*, 1:115–144, October 1981.

[CLE88] A. K. Caglayn, P. R. Lorczak, and S. E. Eckhardt. An experimental investigation of software diversity in a fault tolerant avionics application. In *Proceedings of 7th Symposium on Reliable Distributed Systems*, pages 63–69, October 1988.

[Cri80] F. Cristian. Exception handling and software fault tolerance. In *Digest of Papers, FTCS-10*, pages 97–103, Kyoto, October 1980.

[Dav87] I. J. Davis. A locally correctable AVL tree. In *Digest of Papers FTCS-17*, pages 85–88, July 1987.

[EL85] D. E. Eckhardt and L. D. Lee. A theoretical basis for the analysis of multiversion software subject to coincident errors. *IEEE trans. on Software Engineering*, SE-11, December 1985.

[GAA80] A. Grnarov, J. Arlatt, and A. Avizienis. On the performance of software fault tolerance strategies. In *Digest of Papers, FTCS-10*, pages 251–253, Kyoto, October 1980.

[Hor74] J. J. Horning. *A program structure for error detection and recovery*, volume 16 of *Lecture Notes in Computer Science*, pages 171–187. Springer Verlag, 1974.

[KA83] J. P. J. Kelly and A. Avizienis. A specification oriented multiversion software experiment. In *Digest of Papers, FTCS-13*, June 1983.

[Kan87] K. Kant. Software fault tolerance in real time systems. *Information Sciences*, 42:255–282, 1987.

[KL86] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion software. *IEEE trans. on Software Engineering*, SE-12, January 1986.

[KS85] K. Kant and A. Silberschatz. Error propagation and recovery in concurrent environments. *The Computer Journal*, 28(5):466–473, November 1985.

[LY81] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-Trees. *ACM transactions on Database Systems*, 4:650–670, 1981.

[MM81] D. R. McGregor and J. R. Malone. Design of a robust, simple and highly reliable filestore. *Software Practice and Experience*, 11:943–947, 1981.

[MSR77] P. M. Melliar-Smith and B. Randell. Software reliability, the role of programmed exception handling. *Sigplan Notices*, 12:95–100, March 1977.

[Par77] D. L. Parnas. *The influence of software structure on reliability*, volume 1 of *Current trends in Programming methodology*, pages 111–119. Prentice Hall, 1977.

[Ran77] B. Randell. *System Structure for Software Fault tolerance*, volume 1 of *Current trends in programming methodology*, pages 195–219. Prentice Hall, 1977.

[Sag85] Yehoshua Sagiv. Concurrent operations on B-Trees with overtaking. In *Proceedings of 4th ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 28–37, March 1985.

[Sco84] R. K. Scott. Experimantal validation of six fault tolerant software reliability models. In *Digest of Papers, FTCS-14*, pages 102–107, June 1984.

[Shr78] S. K. Shrivastava. Sequential pascal with recovery blocks. *Software Practice and Experience*, 8:177 –185, March 1978.

[Shr79] S. K. Shrivastava. Concurrent pascal with backward error recovery- language features and examples. *Software Practice and Experience*, 9:1001–1020, December 1979.

[SM85] S. C. Seth and R. Muralidhar. Analysis and design of robust data structures. In *Digest of Papers FTCS-15*, pages 14–19, June 1985.

[SS82] D. P. Siweiorek and R. S. Swartz. *The Theory and Practice of Reliable System Design.* Digital Press, 1982.

[SS85] M. C. Sampaio and J. P. Sauve. Robust trees. In *Digest of Papers FTCS-15*, pages 23–28, June 1985.

[TB85] D. J. Taylor and J. P. Black. Guidelines for storage structure error correction. In *Digest of Papers, FTCS-15*, pages 20–22, Ann Arbor, Michigan, June 19-21, 1985.

[TB86] D. J. Taylor and J. P. Black. A locally correctable B-Tree implementation. *The Computer Journal*, 29:269–276, 1986.

[TMB80] D. J. Taylor, D. E. Morgan, and J. P. Black. Redundancy in data structures: Improving software fault tolerance. *IEEE trans. on Software Engineering.*, SE-6:585–594, Nov. 1980.