A Model for Storage Structures, Encodings, and Robustness

J.P. Black
D.J. Taylor

Data Structuring Group
CS-84-45

December, 1984

A Model for Storage Structures, Encodings, and Robustness

James P. Black

David J. Taylor

Department of Computer Science
University of Waterloo
Waterloo, Ontario
Canada. N2L 3G1

ABSTRACT

We present a model which unifies the treatment of data structure robustness at the storage structure and encoding levels of abstraction. The model provides a sufficiently precise definition of storage structures for robustness analysis to be meaningful, and yet remains sufficiently general that such analyses are useful for widely different encodings of the storage structure. It also permits precise analysis of a large class of combinations of storage structures, based on the interactions between their encodings.

The basic concepts of storage structure robustness are presented in terms of the model, including changes, detectability, and valid state hypotheses. New concepts are introduced for the encoding level, and some results are given relating storage structure and encoding detectability. The utility of the model is illustrated with an example of a combination of three storage structures.

1. Introduction

Over the last several years, it has become increasingly clear that complex computer systems must be tolerant of both hardware and software faults. Anderson and Lee [1] give a comprehensive treatment of the current state of the art; the proceedings of the Fault Tolerant Computing Symposia and of the Symposia on Reliability in Distributed Software and Database Systems should be consulted for more recent work in the field.

Our work on storage structures has been concentrated on exploiting structural redundancy in data to detect and possibly correct errors in storage structures. The techniques involved can be incorporated at different levels of abstraction in a computer system, resulting in a significant degree of tolerance to hardware and/or software faults which damage important data.

In other papers [5,10,11], we have presented many of the basic notions concerning robust storage structures, in which some number of erroneous changes to structural information can be guaranteed to be detected and possibly corrected. Those papers were somewhat informal regarding the underlying definitions of storage structures and encodings, the system state surrounding an instance of a structure, and the conditions under which a general analysis of the robustness of a storage structure would indeed be applicable to the robustness analysis of an actual implementation. In this paper, we not only give a firmer foundation for robustness analyses, but also show how the robustness of complex systems of storage structures can be precisely determined for realistic implementations and error models. Although intended primarily for robustness analysis, we believe the model we present provides some important insight into the differences between the storage structure and encoding levels of abstraction.

This paper assumes very little prior knowledge of robustness theory and practice, although some prior acquaintance would certainly be useful in understanding the motivation behind the model we present. The interested reader is referred to [2,3,4,12] as well as the references cited above. However, we will be defining some of the most basic concepts in terms of the formal model we develop.

Section 2 gives our definition of a storage structure, and defines the notion of an instance of a storage structure and its correctness. Section 3 uses this to define the basic robustness concepts of detectability, correctability, error models, and valid state hypotheses. Section 4 describes the encoding level of abstraction, and its relation to the storage structure level. This is exploited in Section 5 to provide a method for the analysis of systems of encodings and storage structures. Section 6 presents an example and some conclusions.

2. Storage Structures

We identify three levels of abstraction associated with the organisation of data in a computer system: data structures, storage structures, and encodings. For example, a binary tree might be considered to be a data structure, a storage structure for a binary tree might consider the tree to consist of a graph with certain properties, while an encoding of such a graph might organise a set of memory locations to contain data fields, memory addresses, and pointer tags.

A *data structure* thus describes the organisation of data at a high level of abstraction, often by means of the operations which may be performed on the data. Formal specifications of data structures are often in terms of abstract data types [7]. At the storage structure level of abstraction, attention is more often focused on how the data is stored, rather than how it may be operated upon. As such, it is often convenient to adopt some sort of graph model for storage structures, and to consider them largely in isolation from procedures which access them. The designer of a storage structure is often concerned with the classic space-time tradeoff, which we feel should be extended to consider robustness as a third dimension of storage structure design. The encoding level of abstraction concerns the properties of a particular implementation of a storage structure (and hence a data structure).

Our overall philosophy is that careful use of redundancy in storage structures and encodings can be used to complement all the other various techniques normally used to increase software system reliability. With this in mind, we will now present our definition of a storage structure.

A *storage structure* consists of a set of *nodes* connected by *edges*. However, as we wish to model the way structures are stored and used in computer systems, the mathematical notion of a graph as an ordered pair (N, E), with N a set of nodes and E a subset of $N \times N$ representing (directed) edges, requires some modification for our purposes. In particular, we wish to associate arbitrary information with each node, treat this information identically to the edges in terms of "changes" or errors, and be able to access all nodes in the instance from one or more "header" nodes.

Intuitively, a "change" is a fixed-size modification to all or part of a node; changes may or may not be erroneous. Programs which modify a structure do so by making a sequence of changes which are intended to transform one correct instance into another. In this paper, we consider the effects of erroneous changes on encodings and storage structures. Such erroneous changes must ultimately be caused by hardware or software faults, but we do not examine the complex manner in which this may occur.

In view of robustness analyses based on "changes", we will say that a node is a tuple of *components*. Components may be of arbitrary *type*, such as integer, real, or string. Some components indicate edges in the graph; others may be data, or structural information such as counts, identifiers, tags, keys, heights. Note that the edges are parts of nodes, not distinct elements as in the graphs discussed above. All nodes of a storage structure instance are assumed to be accessible by following some sequence of edges from one or more distinguished "header" nodes. (We will reserve the term "root" for trees; in some structures, the root node is different from the header node or nodes.) Finally, each component of a node has a *name* (e.g., "forward pointer"), and a value (e.g., "node 3").

It is important to draw a clear distinction between the storage structure considered in general, and an *instance* of it. The storage structure itself consists of a set of instances; instances belonging to the set can be described by a predicate which correct instances must satisfy. The use of a predicate description is a convenient way to avoid having to enumerate the possible instances. In this paper, we will say that an instance is *correct* if and only if the given predicate is true when applied to the instance. We would normally give the predicate as a list of *axioms,* the predicate simply being the conjunction of the axioms. This approach to storage structure description is elaborated in [2].

We require a notation for storage structure instances. We will denote a storage structure instance S by

$$S = ssi(SS; N; H; I),$$

where SS represents the storage structure description and component names; N is a *node space* or set of node names in which the instance may be found; H is a tuple of header nodes specified by node names from N; and I is an *instance mapping.* The instance mapping takes a node name and a component name, and returns the value of the named component of the node. More formally, if CN is the set of component names of SS, and CV the union of the ranges of components in SS, then we have

$$I: N \times CN \rightarrow CV.$$

For an edge named en stored in a node nn, $I(nn, en)$ is a node name in N. In general, N contains nodes which are part of the instance (accessible by a path from a node in H), as well as nodes which are not.

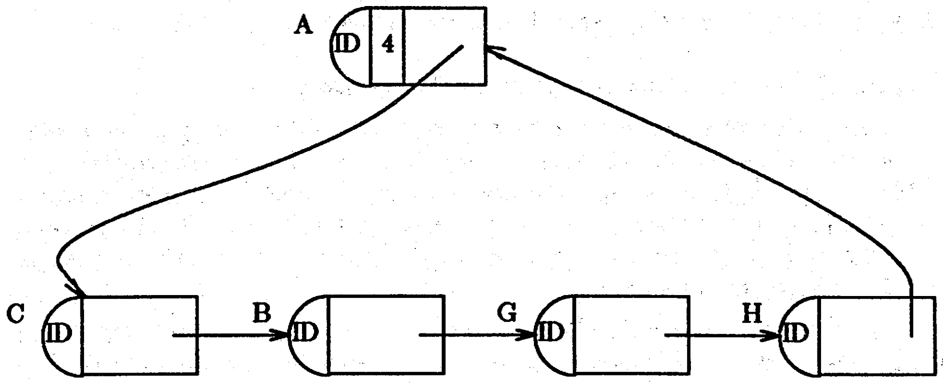
In many cases, it is convenient to think of I as a partial function, in that not all nodes contain every type of component, and in that some nodes of N are not part of the instance. As an example, Figure 1 shows two forms for the instance mapping of

$$S = ssi(Single\text{-}Linked\_List; \{A, ..., H\}; (A); I).$$

The list consists of a header node, A, and four data nodes C, B, G, and H. Nodes D, E, and F also belong to the node space, but are not part of the instance. Every node in the instance contains an identifier component, and an edge component (pointer) to the next node in the list. In addition, the header contains a count of the number of nodes on the list. As examples, the instance mapping shown in the figure has

$$I(A, count) = 4, \text{ and}$$
$$I(B, next) = G.$$

Note that the figure does not define the instance mapping I for nodes D, E, or F, nor for the count component in any node other than the header. Informally, the header node is of a different type from the others: it alone contains a count component.



	id	count	next
A	ID	4	C
B	ID		G
C	ID		B
D			
E			
F			
G	ID		H
H	ID		A

Figure 1. Two forms for an instance mapping

More formally, the *type* of a node is a tuple of component names, indicating what information is stored in the node in a correct instance. In storage structures which have more than one type of node, any procedure manipulating instances of the structure must be able to determine the type of each node it encounters. If this determination is subject to error, the instance mapping may well be applied to arguments for which it is intuitively undefined. The actual values returned for "undefined" arguments will most likely be related to the underlying encoding being used. Hence, we require that I be total in order to perform robustness analyses and to model realistically the operation of procedures manipulating storage structures subject to error.

One further point concerns identically-named components in different node types. In order to make the instance mapping unambiguous, we will assume that all identically-named components appear in the same tuple position in all node types, or that component names are qualified by node type where required.

In the example of Figure 1, the single-linked list nodes all include a component called "id", whose value is assumed to uniquely identify the type of the node and the instance in the node space to which the node belongs. Such identifier components provide useful redundancy which often improves the detectability and correctability of a storage structure. An *identifier component* in a node of a correct instance has a value which is sufficient to determine the node type and the instance to which the node belongs. They thus serve to detect changes of node type and the presence of nodes not part of an instance.

3. Error Models and Storage Structure Robustness

The objective in designing a robust storage structure is to guarantee that some number of erroneous changes to any instance of the structure can be detected, and possibly corrected. In order to determine whether a given structure is robust, a formal analysis of its properties must be undertaken. In this section, we will discuss how the storage structure model of the previous section is used as a basis for robustness analyses.

We first discuss the basic notion of a change to a storage structure instance, which is used to define the distance between two instances and the detectability of a storage structure. As this results in a very pessimistic value for detectability, one of a number of "Valid State Hypotheses" may be made in order to calculate a more realistic value for detectability. Notions of storage structure correctability follow directly from the definitions of distance and detectability.

At the storage structure level, a *change* modifies the value of exactly one component in a node space, that is, the value of $I(nn, cn)$ for one (nn, cn) pair. In structures with more than one node type, the effect of a single change is more complicated. As the instance mapping can be interrogated for a component which does not logically occur in a particular node, any single change to a component appears to change the corresponding component in all other node types. These corresponding components occur at the same position in all tuples defining the various node types of the storage structure.

For example, a CTB-tree [6] contains three types of node: the header node, leaf nodes, and branch (interior) nodes. If the root pointer in the header node, the "thread" pointer in leaf nodes, and the first tree pointer in branch nodes all happen to occur in the same tuple position, a single change to $I(H, root)$ also appears to change $I(H, thread)$ and $I(H, P[1])$. For a given node H, however, only one of these components "actually" occurs in H in a correct instance. At this level, components are assumed to be integers, reals, strings, edges, etc.; the size of a component is known only to be the same as the size of a change. In Section 4, we show how to analyse particular encodings of a storage structure for varying component sizes and implementations. It is also possible to consider a model for errors in which a single "macro change" arbitrarily alters the instance mapping for an entire node. We will not discuss this here for reasons of brevity, and because the development is quite similar to that for single component changes. For more detail, the reader is referred to [2], Chapter 4.

Let $S_1 = ssi(SS; N; H; I_1)$ and $S_2 = ssi(SS; N; H; I_2)$ be two storage structure instances in the same node space, with the same headers, but with different instance mappings. Then the *distance* between the two instances, $d(S_1, S_2)$, is the smallest number of changes to I_1 which can make it identical to I_2. Note that this is the same as the number of changes to I_2 which can make it identical to I_1, and that none of the changes is necessarily to a node which is connected to H. We will say that S_1 and S_2 are *indistinguishable* if their instance mappings are identical on the set of nodes connected to H. Given the distance metric, d, and the notion of indistinguishability, we will say that a storage structure is *k-weak-detectable* if, for any correct S_1 and S_2 which are not indistinguishable, $d(S_1, S_2) > k$. This form of detectability is "weak" in that it assumes nothing about the contents of nodes not connected to the headers, and hence is a pessimistic value. Unless stated explicitly, we will only use the term distance (and the metric d) with respect to distinguishable instances.

The definition of detectability relies on the ability to decide whether an instance is correct. As mentioned above, this correctness criterion is equivalent to deciding whether an instance satisfies the set of axioms describing the storage structure. Although our intent is not to discuss the complexity of error detection in storage structures, almost all robust storage structures known to the authors have linear-time detection procedures which are easily constructed from the storage structure axioms. We currently know of no robust storage structure requiring more than $O(n \log n)$ time to detect errors in an n-node instance.

The definition given above for weak detectability is unnecessarily pessimistic, as it assumes nothing about the contents of nodes outside the instance but within the same node space. In this case, any calculation of detectability must assume a worst case: nodes outside the instance contain precisely the worst possible values, so that an "intelligent adversary" can make a small number of changes to nodes in the original correct instance, in order to transform it into a different correct instance. The new correct instance may contain nodes which were originally outside the instance, but which have been easily incorporated into it due to the fortuitous values which they contained. If this "worst case" analysis is to be made less pessimistic, some assumption must be made about the contents of nodes outside the original instance. Such an assumption is termed a *Valid State Hypothesis* (VSH).

One VSH which might be made is that no node external to the correct instance contains an identifier component whose value is valid for this instance. While this would only impose restrictions on identifier components, we will make a slightly stronger VSH in order to be compatible with the original definitions made in [12]. Our Valid State Hypothesis will be:

1. No node outside the instance (i.e., not connected to the header(s)) appears to have a valid identifier component for this instance, and
2. No node outside the instance appears to have an edge component whose value is a node name in the instance.

A correct storage structure instance in a node space which satisfies this VSH will be termed *valid*. Neither of the restrictions of the VSH absolutely prohibits the storing of such values elsewhere in the node space; the only requirement is that they not be stored in component positions which could be interpreted as identifier or edge components for this instance. Note that the storage structure model uses a node space identical to the range of edge components; implementations in which pointer fields may contain values which do not necessarily correspond to node boundaries are incorporated by considering the encoding level of abstraction in the next section.

Given the VSH made above, we will say that a storage structure is *k-detectable* if, for any two distinguishable, correct instances S_1 and S_2, such that the node space surrounding S_1 satisfies the VSH (S_1 is valid), $d(S_1, S_2)>k$. Note that only one of the instances is required to be valid. This reflects, for example, the operation of an erroneous update routine which, although given a correct and valid instance, may make some erroneous changes to the instance which leave identifier values or edges into the instance in nodes external to it.

A yet more optimistic definition of detectability, called *absolute detectability,* assumes that both S_1 and S_2 satisfy the VSH; equivalently, an intelligent adversary seeking to make undetectable modifications must "clean up" the surrounding node space.

Given these definitions, analysing the detectability of a storage structure can proceed along one of two lines. For a particular structure, it may be possible to argue its detectability directly from the storage structure axioms. When this is feasible, it results in the determination of the largest value of k for which the structure is k-detectable. Such an analysis is often based on a partitioning of the set of undetectable modifications into those which do not change the set of nodes in the instance, those which replace some nodes with previously foreign nodes, and those which change the number of nodes in the instance. The other approach is to attempt to use some general bounds on detectability which are easily determined from properties of the storage structure. More information on calculating detectability may be found in [2,11,12].

The most general bounds on detectability are based on the notion of a *determining set* of components. A storage structure is *k-determined* if any instance of it contains k disjoint sets of components (determining sets), and there exists a "reconstruction algorithm" for each set which can entirely reconstruct the instance, examining only components belonging to the set. Each reconstruction algorithm may use any means at its disposal, but may only interrogate the instance mapping for components in the corresponding set.

We will discuss one easy result based on determining sets: a storage structure which is k-determined is $(k-1)$-detectable. No set of $k-1$ or fewer changes can alter all determining sets. If the k reconstruction routines are invoked, one of them must recreate the original, correct instance; the error(s) can be detected by comparing this instance against the actual erroneous instance. For example, a double-linked linear list is 2-determined: the two sets consist of the forward and back pointers respectively.

For completeness, we will briefly discuss the correctability of robust storage structures. A storage structure is termed *r-correctable* if, given an erroneous instance of the structure assumed to contain at most r errors (changes), there exists a procedure which can recreate the original, unchanged instance given only the (erroneous) instance mapping. Roughly, $2r$-detectability is sufficient for r-correctability, as in the case of binary codes [8]. However, as the set of all possible instances of a storage structure is much less structured than the set of all possible code words in the binary code case, it may be quite difficult to find an efficient correction procedure. Nevertheless, we have shown in [2] that there exist linear-time multiple error correction procedures for several generally useful storage structures. Correction procedures are also discussed in [5] and [9].

4. The Encoding Level

The storage structure model of robustness developed in the two previous sections allows storage structures to be designed and analysed with robustness in mind. The process of implementing a robust storage structure results in an *encoding* of it. Depending on many external considerations, the encoding may well introduce complexities and extra storage structure inter-relationships which require careful evaluation before the robustness of the storage structure as an abstraction can be claimed for the actual encoding level. Consider the following list of problems.

1. How do changes at the encoding level involving 32-bit words or 512-byte disk sectors (for example) relate to the single component change or macro change models at the storage structure level?
2. What if several components are packed into a single "location" affected by an encoding change?
3. What if erroneous pointer values can point to locations which are not node boundaries?
4. What if one encoding depends on another? An example is when pointers in a data base system are encoded as a pair (*page no.*, *record no.*), and the correspondence between logical page numbers and physical disk addresses is itself maintained in some encoding of a storage structure. A similar example is the splitting of logical records across physical page boundaries, but with logically contiguous pages not necessarily physically adjacent.
5. What is a meaningful measure of the robustness of a structure such as a double-linked list, if each node in the list also contains a header for some other structure, or even an entire storage structure instance?
6. To what extent is it appropriate to use a storage structure detection or correction procedure for a particular encoding?

Our solutions to these problems involve careful definition of the relationships between the encoding and storage structure levels, and analyses of the interactions between encodings. The results which we derive permit robustness analyses of encodings for an arbitrary fixed-size change to the stored representation of the data, and for quite general combinations of structures. As the results make use of robustness analyses for storage structures, they permit software system designers to use robust storage structures in an "off the shelf" manner, in a way which can guarantee the robustness of the resulting operational software system.

In the remainder of this section, we first define encodings more precisely, and then consider relating encodings and encoding changes to the storage structure level. This is followed by a formalisation of some possible relationships between encodings and encoding instances in an operational system, and a brief discussion of the main results for encoding-level robustness.

We will define an *encoding* to be a triple

$$E=(F, T, A).$$

F is an *interpretation function,* T a *translation function,* and A a set of *addresses*. The interpretation function corresponds to the storage structure instance mapping, the translation function relates bit strings to storage structure component values, and nodes of an encoding instance are named by addresses. An encoding instance is encoded in a *data space,* which consists of a set of *locations*. Each location is a fixed-length string of bits, with the size of a location being the same as the size of an *encoding change* used for a robustness analysis of the encoding. We will refer to the contents of a data space D by $contents(D)$, and to the contents of a location loc by $contents(loc)$.

Then an encoding instance s of a storage structure SS will be denoted by

$$s = ei(SS; D; H; E),$$

where D is the data space containing s, H a tuple of header addresses belonging to A, and $E=(F, T, A)$ the encoding used for s.

The interpretation function, F, of an encoding uses the contents of the data space D to return a (*bit string, type indication*) pair which corresponds through the translation function T to a component value. Such a (*bit string, type indication*) pair will be termed a *field.* More formally, we have

$$F\colon A \times CN \times \{contents(D)\} \rightarrow \{fields\}.$$

Thus, given an address, a component name, and the contents of the data space, F returns a field. In particular, as addresses are the encoded versions of edges in the structure, they are fields. We will also assume that F may return one or more fields denoted *error*, in the case of incorrect or inconsistent arguments.

On a component by component basis, the translation function T is used to transfer between the encoding and storage structure levels:

$$T\colon \{fields\} \rightarrow \{component\ values\}.$$

T will be assumed to be "invertible", in that given a component value and a type indication, T^{-1} returns a field. Error fields are translated into components also denoted *error*. Thus, T specifies the correspondence between bit strings of the encoding instance and the abstract values (such as node names, integers, strings, ...) of the storage structure components. When convenient, we will extend T to take a set or tuple of fields, and to return the corresponding set or tuple of component values.

Our aim in defining all of this formalism is to be able to relate storage structures and storage structure instances to encodings and encoding instances. The essence of this relationship is that from an encoding instance $ei(SS; D; H; E)$, it is possible to recover the corresponding storage structure instance with the use of the interpretation and translation functions. This instance is

$$ssi(SS; T(A) - error; T(H); I),$$

where, for a node name nn and a component name cn,

$$I(nn, cn) = T(\ F(T^{-1}(nn, address), cn, contents(D))\ ).$$

Intuitively, the node space is constructed from addresses in A for which T does not return *error*; the header nodes of the storage structure instance correspond to the header addresses of the encoding instance; and the instance mapping translates node names into addresses, retrieves the field corresponding to the desired component, and translates this into an abstract component value. (For simplicity, we will often write $ssi(SS; T(A); T(H); I)$, ignoring the possibility of *error* being returned by T.) Notationally, we will use a lower case identifier, such as s, to refer to an encoding instance, while the corresponding storage structure instance will be denoted by the same identifier, S, in upper case.

Note that the storage structure node space depends entirely on the set of addresses for which the translation function does not return *error*. The larger the set, the more nodes and possibilities for invalidity at the storage structure level. For example, if pointers are encoded as byte offsets in a file, and nodes are encoded into a contiguous sequence of bytes, each possible value for a pointer field has an image under T in the node space of the storage structure instance. Most of these nodes would not be part of the instance, nor even part of the intended node space; in such cases, the validity of a robustness analysis relies to a critical extent upon the Valid State Hypothesis being satisfied.

Figure 2 depicts the conceptual correspondence between storage structures and encoding level concepts more graphically.

Encoding		Storage Structure
Addresses (Data Space)	$\xleftrightarrow{T}$	Node Names (Node Space)
Header Addresses	$\xleftrightarrow{T}$	Header Node Names
Fields	$\xleftrightarrow{T}$	Components
Component Names	$\longleftrightarrow$	Component Names
Interpretation Function	$\xleftrightarrow{T, F}$	Instance Mapping

Figure 2. Encoding and Storage Structure Levels

Before developing the results for systems of structures in the next section, we give an example of a storage structure instance, its implementation in a data space, and a tabular form of the storage structure instance corresponding to the contents of the data space. We will use the single-linked linear list of Figure 1, and assume that each field in a node is one location in size. In view of this, we can implement this instance using fixed-length records of three locations each. We will assume that the data space, D, is exactly the size of 8 records, with addresses from the set $\{0, \ldots, 7\}$ corresponding to relative record numbers in D, and that $contents(D)$ are

0:	5	4	2
1:	5	12	6
2:	5	5	1
3:	6	9	8
4:	5	0	0
5:	10	0	100
6:	5	0	7
7:	5	15	0

In order to completely specify the single-linked list encoding instance, we will assume that the interpretation function maps the identifier component onto location 0 of a record, the count or data component onto location 1, and the "next" edge component onto location 2. The translation function maps addresses in [0..7] onto [$A..H$] respectively, 5 into ID (the valid identifier component value for this instance), and represents the count as a standard binary integer. Denoting the encoding by E, the encoding instance in the data space is

$$s = ei(\textit{Single-Linked\_List};\, D;\, (0);\, E).$$

The complete storage structure instance mapping of

$$S = ssi(\textit{Single-Linked\_List};\, T(\{0, \ldots, 7\});\, T((0));\, I)$$

is the following:

Node	id	count/data	next
A:	ID	4	C
B:	ID	12	G
C:	ID	5	B
D:	error	9	error
E:	ID	0	A
F:	error	0	error
G:	ID	0	H
H:	ID	15	A

Note that this storage structure instance does not satisfy the valid state hypothesis: node $E = T(4)$ contains both the appropriate identifier value, and a pointer to the header $A = T(0)$. However, none of nodes D, E, or F are part of the instance.

The definitions given above imply that any one data space may contain many related and unrelated encoding instances, and that each encoding for which an instance occurs in the data space may have a different set of addresses as well as node names to refer to its own nodes. The interpretation function may be arbitrarily complex, using other supporting encodings and encoding instances to construct the field returned for a given component. Together, the interpretation function F and the translation function T allow the abstraction of the storage structure instance from the unstructured set of locations which is the data space. The purpose of the next section is to show how it is possible to perform robustness analyses based on encoding changes to locations in the data space.

5. Robust Implementations of Storage Structures

The main difficulty which we are trying to solve can be summed up in the term "change multiplication": a single change to a location in a data space can appear to cause a very large number of changes to those storage structure instances which can be abstracted from encoding instances in it. This can be due to component packing within a (large) location, as well as to "dependencies" between instances, such as the encoding of pointers using page numbers which themselves must be interpreted via some encoding which supports the abstraction of a file as a sequence of pages.

The following three results can be used when the relationship between the encoding and the storage structure is relatively straightforward.

1. Let $s = ei(SS; D; H; E)$. If SS is $k*j$-detectable, and any encoding change to D results in at most k changes to S, then s is j-encoding-detectable. (Recall that S is the storage structure instance corresponding to s.)
2. If SS is j-macro-detectable (any set of errors involving j or fewer nodes can be detected), and any single encoding change to D results only in changes to one node in $T(A)$, then s is j-encoding-detectable.
3. If no single encoding change modifies more than one determining set in a k-determined storage structure, then the encoding is $(k-1)$-encoding-detectable.

As an example, Figure 3 shows part of a linked list structure, in which pointers are encoded as (*page no.*, *record no.*). Let us ignore for the moment any lower-level encodings supporting the page abstraction, assume each page always contains a 64-entry record index, and that all components shown are stored in single locations. If a 2-detectable double-linked list is used, what is the encoding detectability of the result? The only location changes which are multiplied are those to record index entries. If these changes are viewed as changing both of the two incoming pointers to a record on the list, Result 1 gives 1-encoding-detectability. However, such a single encoding change can also be viewed as a macro change, affecting only one node of the storage structure. As a slightly different implementation of a double-linked list can be made 2-macro-detectable, Result 2 gives 2-encoding-detectability. The viewpoint adopted may well affect the (lower bound on) encoding detectability obtained from the analysis: this is one aspect of a more general tradeoff between the effort required for a detailed analysis, and the tightness of the resulting bound.

Result 3 gives an interesting perspective on pointer tagging, under the assumption that the tag associated with a pointer often reflects the membership of that pointer in one of a number of determining sets. If this is the case, encoding the tag in the same location as the pointer implies that a single encoding change can alter both the pointer and its membership in a determining set. On the other hand, encoding the tag of a pointer in the same location as, say, the previous pointer in the same determining set would allow Result 3 to be applied, possibly increasing the encoding detectability obtained. However, this may require an increase in the total number of tag components stored.

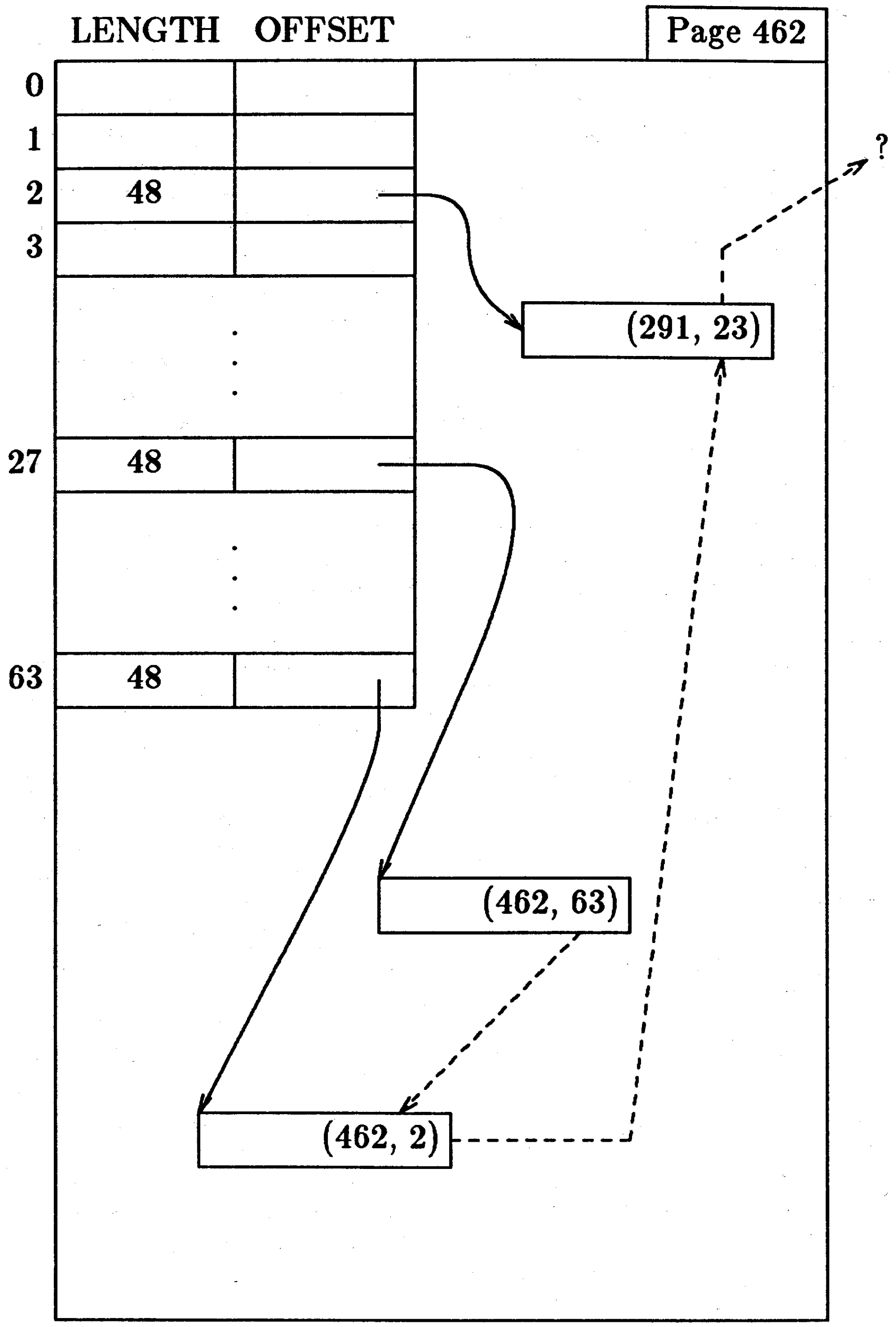


Figure 3. Page numbers and a linked list

Up to this point, we have not given a formal definition of "encoding detectability", relying instead on the reader's intuitive understanding. In order to define the concept properly, and to have a precise idea of dependencies between encodings, we need to introduce the following ideas.

Let $c=(nn, cn)$ be a component of $S=ssi(SS; T(A); T(H); I)$, with $I(c)=x$. Define the *dependency set* of c, denoted $dep(c)$, to be the set of all locations such that changing that location makes $I(c)\neq x$. Inversely, with each location in a data space, we may associate those components of S which are dependent upon it. Thus, we will say that a location *supports* zero, one, or more components of S according to the number of components depending upon it. The set of all locations supporting one or more components of S will be denoted $support(S)$. We note in passing that this set is exactly the union of the dependency sets of all components of S.

Given the supporting set, $support(S)$, of a storage structure instance, it is useful to partition it according to the number of components supported by the locations. Define $kernel(S)$ to be the set of locations supporting exactly one component in one or more node types of the structure, and $context(S)$ to be the set of locations supporting more than one component in some node type of S. We will say that an encoding instance s (and hence the encoding itself) *depends* on another encoding instance t (or encoding) if some location in $context(S)$ also belongs to $kernel(T)$. When $context(S)$ is empty, s is *independent;* an encoding is independent if all possible instances of it are independent. The encoding of the single-linked linear list in the previous section is independent: no encoding change affects more than one component in any node type of an instance.

Figure 4 reproduces Figure 3, with the kernel and context of the double-linked list indicated for that part of the list. The supporting set of the linked list storage structure instance is all the marked locations on this and other pages containing records of the list.

It is important to note that this simple partitioning of the supporting set into kernel and context is somewhat coarse, as more subtle relationships between locations and components can be identified. For example, $context(S)$ is intuitively most meaningful when all the locations in it are part of some supporting structure, as illustrated by the record index example. However, locations which pack together fields for more than one component also appear in the context; such cases may well make a formal analysis using the dependency relationships difficult, as the encoding instance in question may be neither independent nor dependent on some other instance. As a further conceptual complication, locations may fortuitously belong to $kernel(S)$ because they only happen to support a single component of S, whereas the same location supports multiple components of other instances, and is conceptually "part of" some distinct structure. Nevertheless, the simple partitioning of the supporting set into kernel and context is sufficient for our needs.

The results which follow use the dependency relation to determine the encoding detectability of an instance based on those of instances on which it depends. In general, the graph of this dependency relation may be cyclic. In such cases, it may not be possible to construct a detection procedure for the encodings from the storage structure detection procedures, due to the arbitrary change

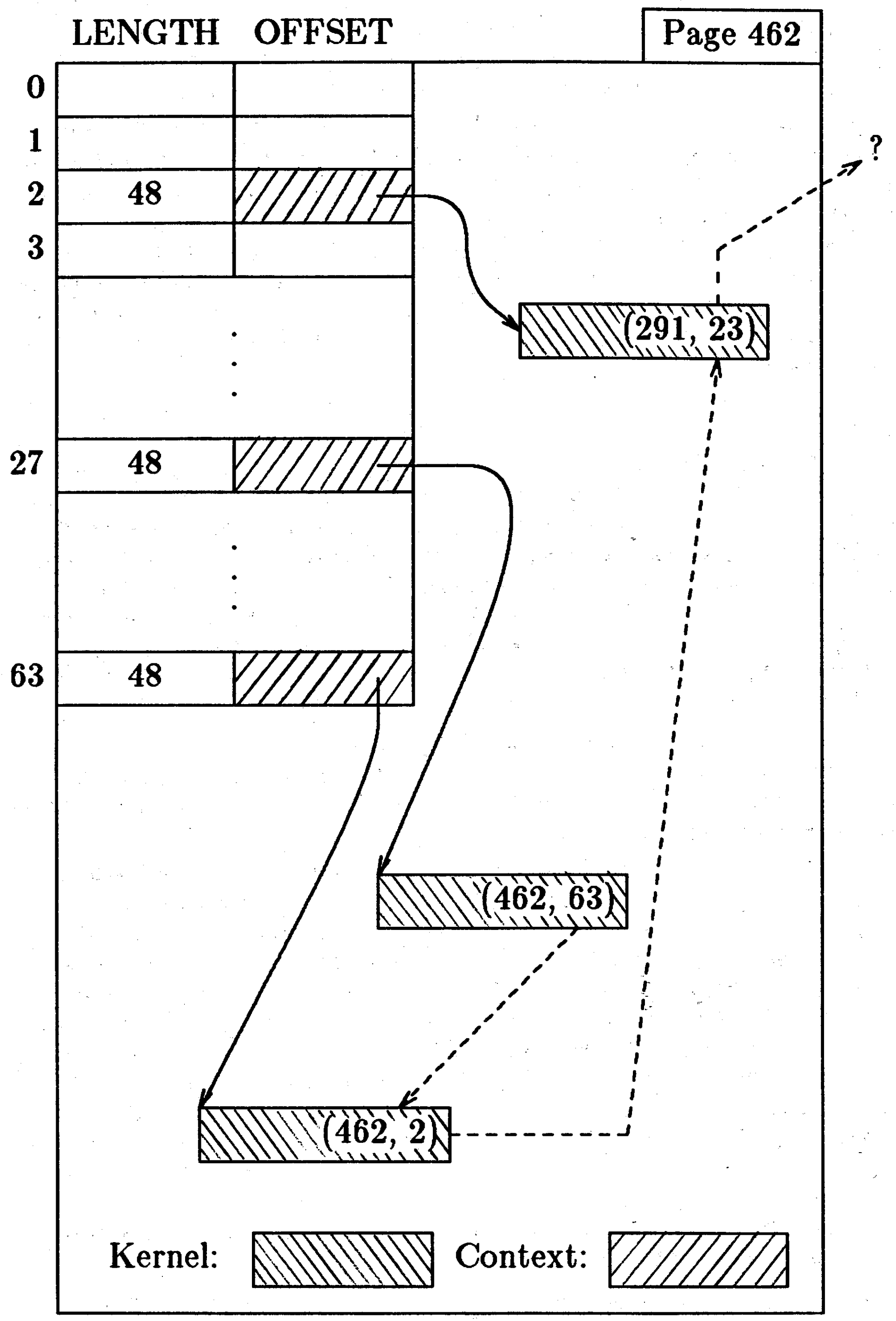


Figure 4. Kernel, context, and supporting set

multiplications which may occur. Two possible means of resolving the difficulty are to find a partitioning of the data into storage structures which removes the circularity, or to devise an *ad hoc* detection procedure which obviates it. In practice, cases of cyclic dependency are expected to be rare, especially when a "reasonable" hierarchical system design is followed.

In order to define encoding detectability precisely, let s, and $t_1, ..., t_n$ be encoding instances whose corresponding storage structure instances $S, T_1, ..., T_n$ are correct and valid, and let s depend only on $t_1, ..., t_n$. Consider a set of j encoding changes which appears to transform $S, T_1, ..., T_n$ into $S', T'_1, ..., T'_n$. Then s is *j-encoding-detectable* if any set of j or fewer encoding changes applied to $support(S)$ causes at least one of $S', T'_1, ..., T'_n$ to be incorrect, or S and S' to be indistinguishable. We will apply the term both to encoding instances, and to their corresponding storage structure instances for the given implementation. Generalising, we will say that an encoding is j-encoding-detectable if every instance of it has that property.

We make the following observation.

4. An independent encoding of a j-detectable storage structure is j-encoding-detectable. This follows directly, as an independent encoding has no context.

Another simple result is the following.

5. Let $SS_1, ..., SS_m$ be storage structures with detectabilities $j_1, ..., j_m$, and let $j = \min(j_1, ..., j_m)$. Then any set of instances of these storage structures which are in the same data space and whose encodings are independent is (at least) j-encoding-detectable.

This latter result applies both to "unrelated" set of instances occuring in the same data space, and to *independent composition,* such as double-linked list of CTB-trees, in which each list node contains a header for a CTB-tree. Formally, each CTB-tree encoding instance is independent from the double-linked list, as no encoding change to the list can change a component of one of the trees. Even if some set of errors to the list caused one of the trees to become inaccessible, that tree would be unchanged and still correct. (More formally, its instance mapping has not changed.) The intuitive dependency of the CTB-tree instances on access paths to their headers is nevertheless reflected in the minimum expression of the result.

The main result based on dependency defines one way in which the encoding detectability of a structure can be calculated from its storage structure detectability and the encoding detectabilities of instances on which it depends.

6. Let SS be a j-detectable storage structure and $s = ei(SS; D; H; E)$ an encoding instance depending only on encoding instances $t_1, ..., t_n$. Let t_i have encoding detectability k_i. If every location in $context(S)$ is also in the supporting set of some T_i, then s is $\min(j, k_1, ..., k_n)$-encoding-detectable.

Let $p = \min(j, k_1, ..., k_n)$, and consider a set of p or fewer encoding changes to the data space. If all of these changes are to affect s, then they must be to locations in $support(S)$, which itself consists of $context(S)$ and $kernel(S)$. As every location in $context(S)$ belongs to the supporting set of some T_i, and as the T_i are individually p-encoding-detectable by assumption, any set of p or fewer changes to $context(S)$ will make at least one of the T_i incorrect. If no changes are made to $context(S)$, then all must be to $kernel(S)$. Any set of p or fewer changes to locations in $kernel(S)$ appears as at most p changes to S, which would leave S incorrect as it is p-detectable.

6. Example and Conclusions

In order to illustrate the ideas in previous sections, we will give a brief example showing how design and analysis of a robust system of storage structures might proceed. We will consider an application which wishes to maintain a linked list of keys (perhaps with other data) in a file on external storage. The reliability specifications for the application require 2-encoding-detectability of the keys and the list structure itself, with a location defined to be a 32-bit word on external storage.

The abstraction of a file, to be provided by the file system, is that of an unstructured byte sequence of arbitrary length, with random access by relative byte addresses within the file. This abstraction is created by allocating an appropriate number of 512-byte physical disk blocks, and associating with these a tree structure used to translate relative byte addresses into disk block addresses and offsets.

A natural decomposition of the problem suggests the implementation of three distinct robust storage structures: one for the keys, one for the list, and one for the tree of physical block addresses supporting the file abstraction. This decomposition implies a dependency hierarchy of three structures. The list depends on the file tree, as a change to a physical block address can arbitrarily alter the contents of a logical block of the file, with arbitrary effects on the list. The key structure can conveniently be viewed as a "Fixed-Length List", with the length a parameter of the storage structure. (This storage structure is similar to the Robust Contiguous List described in [6], except that the number of keys in the list is fixed by a parameter of the structure, with the intent that it be stored in an encoding on which the list is dependent.) This implies dependency of the key array on the list structure implementing the logical contiguity of array elements.

One solution for the tree structure is some type of 2-detectable perfectly-balanced tree. In this case, interior nodes only need to contain pointers, as the relative byte address whose disk location is being sought implies the pointer to be followed at each level of the tree. At the leaves, where physical block addresses are stored, Robust Contiguous Lists can be used to protect their values. The tree can be chained and threaded as a CTB-tree [6], with the difference that all nodes are full except those on the rightmost path from the root. We will assume the encoding of this tree to be 2-encoding-detectable, ignoring possible dependencies of it on the structure used to find files and manage physical disk block allocation.

On this basis, a straightforward implementation of a double-linked list may be used within the file. Each list node contains one key and one difference field used for the higher level robust key array. The robust key array consists of a number of key and difference pairs, related as in a Robust Contiguous List. For a fixed number of data keys greater than one, such a list is 2-detectable (see [2]). In this dependent implementation, the actual number of keys in the array is given by the count component of the double-linked list.

For brevity, we will leave as an exercise to the reader the verification of the fact that the robust key array depends on the linked list, which in turn depends on the file tree. Two applications of Result 3 then give the required encoding detectability of two.

The example shows that the use of robust storage structures can easily be integrated with normal system designs. For the designer who wishes to work in storage/time/robustness space, the results allow him to evaluate the robustness of alternate designs, storage structures, and combinations. For the analyst who wishes to determine the robustness of a given system of data structures, the results allow him to decompose and structure his analysis in a manner which reflects the "natural" structure of the system.

Taken together, these results make it possible to determine the encoding detectabilities of a large class of complicated storage structures, including independent composition and dependent implementation. They permit a hierarchical approach to analysing combinations, as well as to synthesising software systems to meet a specified reliability criterion. The underlying model formalises the relationships between storage structures, encodings, and instances, while keeping the concept of a storage structure sufficiently abstract that widely different encodings may be used to represent different instances of the same storage structure.

None of the results presented here achieve an increase in overall robustness due to combination of several storage structures. Two classes of combinations which may increase robustness have been identified in [2], and are generalisations of the "compound" storage structure originally presented in [11]. However, it appears that stronger results require quite significant restrictions on the structures being combined, and are unlikely to be generally useful.

Using robust storage structures does not exclude using other techniques for coping with faults. For example, stable storage at the disk block level could well be complemented by a robust storage structure to create the abstraction of a file from a set of disk blocks. Together, the two techniques could permit tolerance of both disk hardware faults and of faults in the file system and operating system software. Thus, the use of robust storage structures extends and complements the set of programming and design techniques which can successfully be used to increase fault tolerance in a computing system.

Acknowledgements

This research was generously supported in several ways by the Natural Sciences and Engineering Research Council of Canada, NATO, and the University of Newcastle Upon Tyne. The results presented are essentially those of [2].

References

1. T. Anderson and P. A. Lee, *Fault Tolerance: Principles and Practice,* Prentice-Hall, Englewood Cliffs, N. J. (1981).

2. J. P. Black, *Analysis and Design of Systems of Robust Storage Structures,* Ph. D. Thesis, University of Waterloo, Ontario, Canada (July 1982).

3. J. P. Black, D. J. Taylor, and D. E. Morgan, "A compendium of robust data structures," *Digest of Papers: Eleventh Annual International Symposium on Fault-Tolerant Computing,* pp. 129-131 (24-26 June 1981).

4. J. P. Black, D. J. Taylor, and D. E. Morgan, "An introduction to robust data structures," *Digest of Papers: Tenth Annual International Symposium on Fault-Tolerant Computing,* pp. 110-112 (1-3 October 1980).

5. J. P. Black and D. J. Taylor, "Local correctability in robust storage structures," CS-84-44, Dept. of Computer Science, University of Waterloo (December 1984).

6. J. P. Black, D. J. Taylor, and D. E. Morgan, "A robust B-tree implementation," *Proceedings of the Fifth International Conference on Software Engineering,* pp. 63-70 (9-12 March 1981).

7. J. V. Guttag, "The specification and application to programming of abstract data types," CSRG-59, Dept. of Computer Science, University of Toronto (1975).

8. R. W. Hamming, "Error detecting and error correcting codes," *Bell System Technical Journal* **26** pp. 147-160 (April 1950).

9. D. J. Taylor and J. P. Black, "Principles of data structure error correction," *IEEE Transactions on Computers* **C-31**(7) pp. 602-608 (July 1982).

10. D. J. Taylor, D. E. Morgan, and J. P. Black, "Redundancy in data structures: Improving software fault tolerance," *IEEE Transactions on Software Engineering* **SE-6**(6) pp. 585-594 (November 1980).

11. D. J. Taylor, D. E. Morgan, and J. P. Black, "Redundancy in data structures: Some theoretical results," *IEEE Transactions on Software Engineering* **SE-6**(6) pp. 595-602 (November 1980).

12. D. J. Taylor, *Robust Data Structure Implementations for Software Reliability,* Ph. D. Thesis, University of Waterloo, Ontario, Canada (August 1977).