Local Correctability in Robust Storage Structures

J.P. Black
D.J. Taylor

Data Structuring Group
CS-84-44

December, 1984

Local Correctability in Robust Storage Structures

James P. Black

David J. Taylor

ABSTRACT

Robust storage structures contain sufficient structural redundancy to permit the detection and possibly the correction of errors in structural information. Such structures and their detection and correction algorithms can be incorporated into a computer system at various levels of abstraction in order to increase its reliability and fault tolerance. An important property of some robust storage structures is local correctability, which permits the correction of an arbitrary number of errors, provided the errors are in some sense sufficiently separated from each other. We present examples of such structures, a general definition of local correctability, and a practical approach to designing and exploiting locally correctable robust storage structures.

1. Introduction

There is a growing interest in techniques to assist in the design and implementation of fault tolerant software, that is, software which is capable of detecting failures in hardware or software components, and reacting appropriately. Two examples of software structuring or design techniques which address this problem are recovery blocks [7] and N-version programming [5]. Some systems, such as SIFT [15], use N-modular redundancy of software components on independent hardware to detect and mask hardware failures. The use of atomic actions [8] or transactions [6] normally provides for automatic backward error recovery. At a lower level in most systems, checksums and error detecting/correcting codes [9] have long been used to guard against undetected communications and hardware failures.

The study of robust storage structures [3,12] attempts to suggest ways of using redundancy in data structures to detect and possibly correct errors in them. As such, robust storage structures and their detection/correction algorithms can be used with most of the techniques mentioned above to increase the fault tolerance and reliability of a software system. Our intent is not to discuss storage

structure robustness in general; the interested reader is referred to [2,13,14]. Rather, we wish to discuss a particular property known as local correctability. The results presented in this paper are revisions and generalisations of similar ones in [2].

Intuitively, we will say that a storage structure is locally correctable if it is possible to correct an arbitrary number of errors to structural components of an instance of it, assuming that the errors are "sufficiently separated" from one another. Thus, it may be possible to correct $O(n)$ errors in an instance containing n nodes. Similarly, a storage structure is locally detectable if the insertion of any set of "sufficiently separated" errors results in a detectably incorrect instance. Unfortunately, these characterisations, while capturing the essence of local correctability and detectability, are neither precise enough nor general enough for our purposes.

Before developing formal definitions, we present a brief general background to storage structure robustness. Section 3 discusses various robust storage structures, and informally analyses whether they could be termed locally correctable. This motivates the formal definition in Section 4, which has the disadvantage of not being constructive. This is remedied to some extent in the succeeding sections, which present a practical approach giving sufficient conditions for local correctability. Section 8 contains a discussion and some ideas for further work.

2. Robust Storage Structures

A robust storage structure is one in which some number of errors to structural information is guaranteed to be detectable in any instance of it. "Structural information" is intentionally vague; typical examples include pointers, counts of nodes, height fields in trees, and identifier components. (In a correct instance, the identifier component of each node in the instance has a value which uniquely identifies both the type of the node and the instance to which it belongs.) This notion of robustness relies on that of the correctness of an instance of the storage structure. While we will only use English to describe correct instances in this paper, more formal treatments have been based on an axiomatic description method [2], or the implicit description embodied in the code of a "detection procedure", which is used as a final authority to decide whether a particular storage structure instance is correct [13].

For this paper, we will consider an instance of a storage structure to consist of a set of *nodes* connected by *edges* or pointers. Each node has a unique *name*. A *component* is a pair consisting of a *node name* and a *component name,* and has a value which is obtained by an instance mapping, denoted I, applied to the component. Components which are edges have values which are node names. An instance of a storage structure consists of a tuple of distinguished *header nodes,* denoted H, and all nodes accessible from them (via I) in the *node space* in which the instance is stored.

In order to quantify robustness, it is necessary to assume some particular model for changes (or errors), and their possible effects on an instance, and to make some assumptions about the validity of the node space (or system state) surrounding a correct instance. For the purposes of this paper, we will assume that a single change affects only a single component of structural data in a node of a storage structure instance. Thus, a single change will affect, for example, the value of a single pointer, count, or identifier component. We will assume that the following Valid State Hypothesis holds:

> With respect to quantifying robustness (or "counting errors"), no node external to a correct instance contains information which could be interpreted as a pointer into the instance, and no external node contains information which could be interpreted as a valid identifier value for this instance.

Effectively, the hypothesis allows us to consider an individual instance in isolation, without concern for the contents of nodes external to the instance but within the same node space. For a more detailed discussion of various change models and less restrictive valid state hypotheses, see [2].

Given that single errors affect single storage structure components, it is possible to quantify the detectability of a storage structure. Let the distance between any two correct instances be the smallest number of changes to single components required to transform one instance into the other. If the minimum distance between any two correct instances is $j+1$, say, then the storage structure is *j-detectable,* as any smaller set of changes will leave the structure detectably in error (the detection procedure would reject such a structure, or the instance does not satisfy the storage structure axioms). Similarly, if any set of j or fewer changes can be corrected by some correction algorithm, the structure is *j-correctable.*

3. Examples

In this section, we will examine several robust storage structures in an attempt to decide whether they should be thought of as being locally correctable.

Consider first what is perhaps the simplest structure with non-zero correctability: a double-linked list with identifier components in each node, and a count in the header node containing the number of nodes on the list. A single pointer error can be detected by traversing the list in the forward direction, say, until a mismatch between forward and back pointers is detected. In the presence of a single error, this detection procedure always halts within one node of the erroneous component. This erroneous component can then be corrected by traversing in the reverse direction from the header until corresponding symptoms are observed, at which point enough information is available to correct the pointer in error. (This algorithm was first given in [14], and is also discussed in [12].) As far as local correctability of this structure is concerned, a second pointer error anywhere in the list can render the first uncorrectable, implying that one can never guarantee that two errors, no matter how "separated", can be corrected. Intuitively, double-linked lists are not locally correctable, although single errors can be detected locally.

On the other hand, reorganisation of the redundancy in double-linked lists can produce a structure which is truly locally correctable. The appropriate reorganisation is to make each back pointer point to the second preceding node, rather than the immediately preceding one. In [14], such lists were termed modified(2) double-linked lists, the parameter referring to the distance spanned by the back pointer. In fact, they are also "2-spiral" lists: a k-spiral list has k pointers in each node; one points to the k'th preceding node, and the others point to the first, second, . . ., $k-1$'th succeeding nodes. As was argued intuitively in [11], 2-spiral lists are locally correctable. More generally, a $(k+1)$-spiral is k-local-correctable, as defined below.

For this paper, we will take a 3-spiral as a slightly more general example. Figure 1 shows part of a 3-spiral, drawn to emphasise the "spiral" structure: nodes linked by the back(3) pointers are drawn in the same column, while the forward(1) and forward(2) pointers link nodes from top to bottom, following the left-to-right descending spiral between the columns. A 3-spiral list has three list headers, one of which (H_0) contains a global count of the number of nodes on the list. The forward(1) pointer from H_0 points to the first non-header node on the list. For reasons of uniformity, all header nodes store three pointers which are required to be set, even though some of the pointers must always point to one of the other header nodes.

Consider the following basic step in a (local) correction algorithm. The algorithm has corrected some number of previous nodes in reverse order to that defined by traversing the structure using forward(1) pointers. At each step, it follows a back(3) pointer, and examines the node reached to verify that the two forward pointers there are set to the two nodes last encountered. If this is the case, the algorithm repeats with the next node in reverse order. For example, as the algorithm reaches node 3 from node 6, it checks that pointers in node 3 are correctly set to nodes 4 and 5. If not, diagnosis is performed by attempting to reach the same node by two alternate paths: back(3) forward(2) from the last node considered correct, and back(3) forward(1) from the node which was encountered two steps ago. In the example, the algorithm would attempt to reach node 3 from node 4 via node 1, and from node 5 via node 2. In the presence of a second error in the locality, the algorithm could be faced with a choice between three different nodes as the next node which should be traversed. However, assuming at most two errors in the locality, the values in the forward pointers of the candidate nodes can be used to correct the inconsistency observed initially, and the algorithm can correct any number of sufficiently separated single or double errors appearing further down the list. Figure 2 shows the components used in one step of this algorithm. Such sets of components will later be referred to as substructure instances.

Although the preceding description is couched in terms of a correction algorithm (containing an embedded 2-local detection algorithm), it would also be possible to use a "conservative" detection algorithm which always performed the diagnosis step described above, even if no error was detected initially. Such a routine would be able to halt in the vicinity of an error unless it encountered more than four errors at once. An even more conservative algorithm can achieve 5-local-detectability.

We do not pretend that the discussion above is a formal proof of local correctability: we will return to that in Section 7. Rather, we wish to illustrate two important aspects of local correctability: one is that the notion of 2-local-correctability is essentially that the correction algorithm can guarantee the

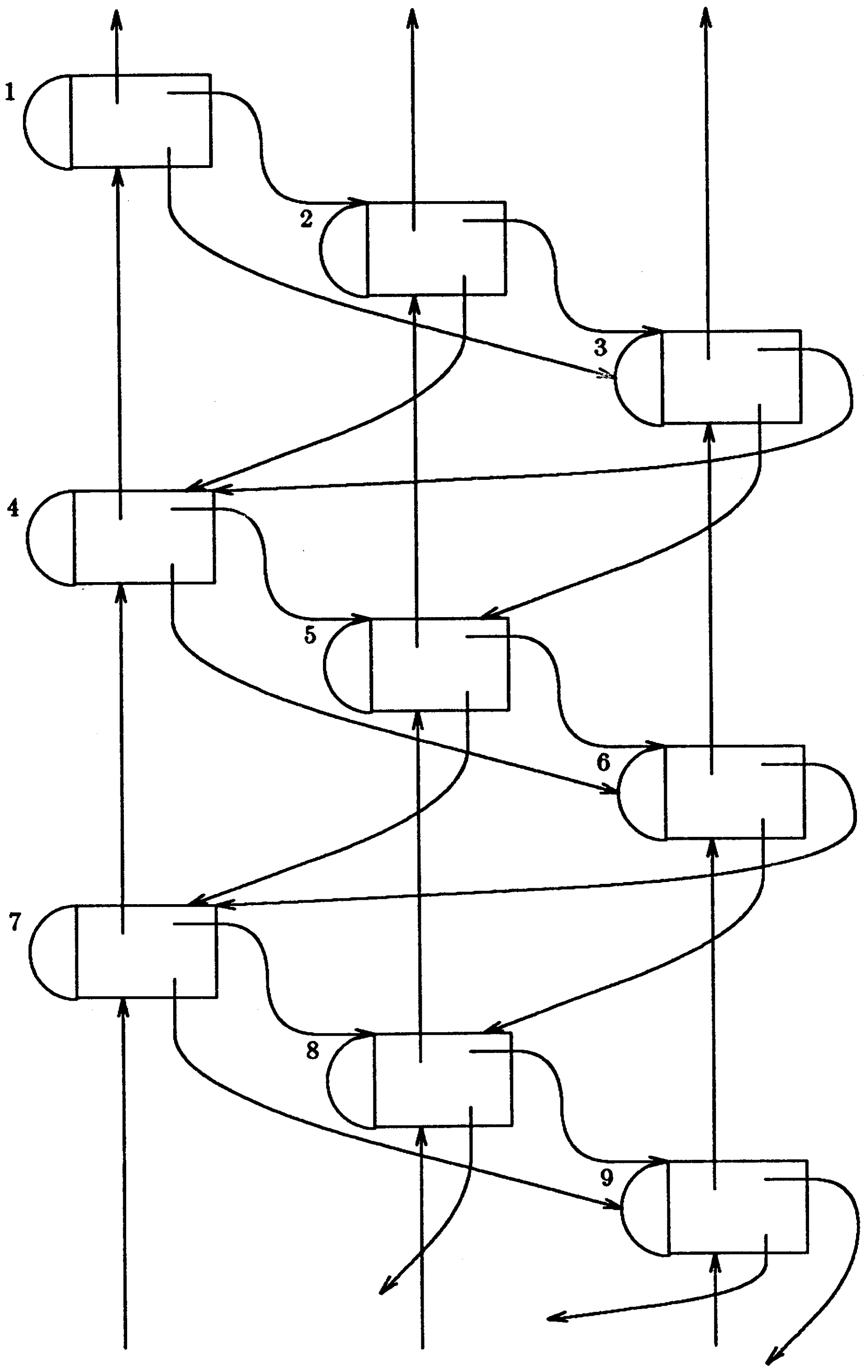

Figure 1. Part of a 3-spiral list

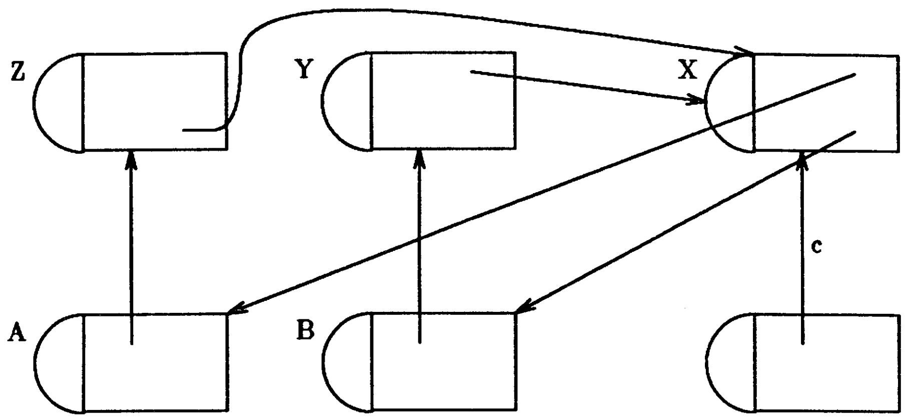

Figure 2. Instance of general substructure, 3-spiral list

correction of any number of errors, providing that it never encounters more than two at once. The other important characteristic which we wish to include in local correctability is that of a constant-sized "locality" around an error, which formalises the concept of the algorithm "encountering" an error.

Other classes of linear lists can be analysed for their local correctability; this is discussed in more detail in [2]. In a sense, a $(k+1)$-spiral has "maximal" local correctability, being both k-correctable and k-local-correctable.

The chained and threaded binary tree, or CT-tree [12], was designed to provide a binary tree storage structure which was 2-detectable and 1-correctable. From the point of view of implementing an algorithm for 1-correction, the principal difficulty is that there is only a loose and obscure relationship between any given tree pointer and the chain and thread redundancy which permits correction of an error to that pointer. This same difficulty in relating tree pointers and chain/thread pointers precludes local detectability and correctability, as the locality of a tree pointer error could include an arbitrary number of chain/thread pointers, and single pointer errors could effectively prevent correction of any other error. A more detailed discussion may be found in [11].

CTB-trees [4] are B-trees [1] incorporating similar chain and thread redundancy, and are much more tractable than chained and threaded binary trees because of their regular structure. In trying to repair a pointer error, the correction algorithm need only scan slightly more than the subtree which was pointed to in order to diagnose and repair the error correctly. (The scan is an in-order scan using either tree pointers or chain and thread pointers.) In this structure, it would be possible to correct multiple errors, provided (roughly) that they occurred in separate subtrees. Thus, the locality associated with an error has a size which depends upon the height of the error above the leaves of the tree. While one could consider this structure to be locally correctable, we will not do so, as we wish to insist upon the requirement for a constant-sized locality.

Figure 3 shows an example of a "linked" B-tree, or LB-tree [10], which is a storage structure designed specifically to be locally correctable according to the definition formalised in the next section. The basic redundancy incorporated is a pointer from each interior or leaf node to its right sibling, and a pointer from each rightmost son to its father. A separate header for each level contains a pointer to the leftmost node at that level, and a pointer to that header is stored in the sibling pointer of the rightmost node. The use of these level headers and associated circular lists provides a path to each node which is edge-disjoint from the tree pointer path to the same node. In a practical implementation of an LB-tree, care must be taken with this variable number of list headers. While not wishing to ignore this problem, discussing it here is beyond the scope of this paper; the interested reader is referred to [10].

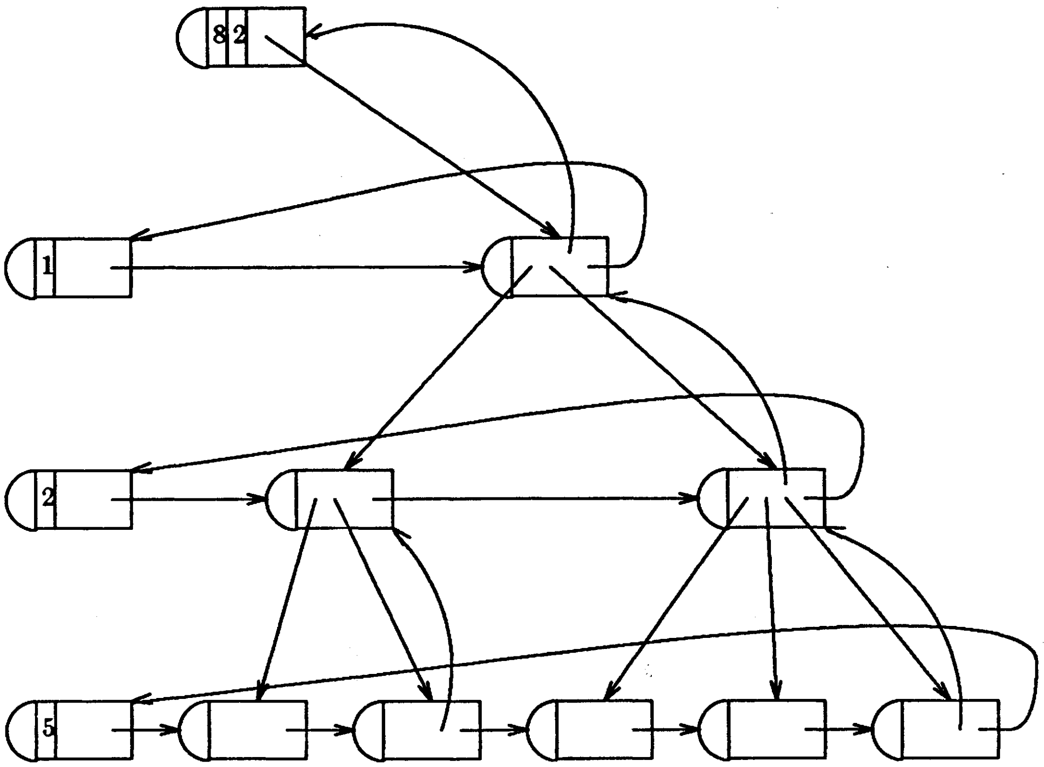

Figure 3. A linked B-tree

In addition to being 1-correctable and 3-detectable, LB-trees are 2-local-detectable and 1-local-correctable. The correction algorithm traverses the structure in breadth-first order, that is, the root, its sons from left to right, their sons from left to right, and so on. In general, the algorithm deals with a part of the structure at each step, as shown in Figure 4. Nodes A and B have already been "validated", and the algorithm is attempting to validate the sibling pointer from B to C. This is achieved by comparing it with the i'th pointer in A. If a discrepancy is observed, diagnosis consists of comparing the pointer from C to D with the $i+1$'th pointer in A. If these latter two agree, the problem is with the

i'th pointer in A; otherwise, the pointer in B must be corrected. Note that the number of storage structure components investigated in the locality is bounded by a constant for this structure. Obviously, the algorithm must deal with several other possible "substructures" during its operation; these correspond to initialising the algorithm at the root of the tree, starting and ending the traversal at each level, and crossing from the subtree rooted at A in Figure 4 to A's sibling.

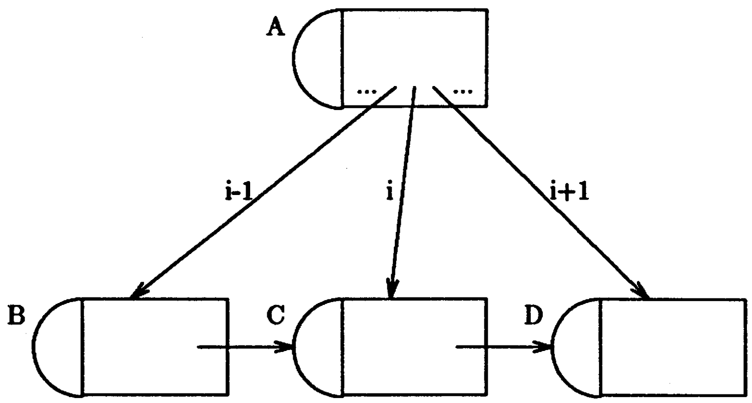

Figure 4. LB-tree general substructure

In the next section, we will make use of the intuitive discussion of these examples of storage structures to motivate our general definitions of local detectability and correctability. This will be followed by a more constructive approach to local correctability based on the notion of substructures.

4. General Definitions of Local Detectability and Correctability

As we have mentioned, we wish to define local correctability of a storage structure to mean that any number of errors can be corrected in an instance of the structure, provided only that the correction algorithm never encounters too many errors at once. Once a satisfactory definition of encountering a set of errors has been made, we will be able to quantify local correctability by the size of the set of errors. Unfortunately, these ideas seem very intimately related to an algorithm to perform correction, rather than to the storage structure itself.

While local correction based on substructures such as those used in the examples above is appealing, it does not appear necessary to restrict local correction algorithms to this "substructure-at-a-time" approach. We would not wish to exclude an arbitrarily complex global algorithm which performed correction subject only to the "separation constraint" imposed upon correctable sets of errors.

Yet, any meaningful definition of separation between components (or errors) seems dependent on some conceptual order of traversal of a storage structure instance. This is particularly true if we wish to restrict local correction algorithms to following pointers from the header(s) of a structure rather than making arbitrary "stabs" at system memory. As a further requirement of the definition, we would wish that it be "fair", in the sense of not making some components or component types "more correctable" than others. For example, given a double-linked list, we wish to exclude an algorithm which will correct any number of back pointer errors by blindly setting back pointers to agree with whatever forward pointer structure the algorithm observes. Finally, we would like the separation required between errors or sets of errors to be bounded by some constant, which excludes CTB-trees from being locally correctable.

With these requirements in mind, we will present our definition of local correctability, which is unfortunately rather more complex than we would have wished. It is based on an abstract model of the operation of local detection and correction algorithms. As defined below, local detectability is equivalent to the existence of a "linearisation function" with certain properties. Intuitively, a linearisation function produces a sequence of components which is guaranteed to terminate in the vicinity of an erroneous component unless the vicinity contains too many errors. Thus, the linearisation is used to measure the distance between errors. Local correctability is then defined in terms of an algorithm which can correct at least one error in the linearisation of an erroneous instance.

We define a *linearisation* of a (possibly incorrect) instance of a storage structure to be a sequence of components from the instance, that is, of components in nodes connected to the header(s). In general, a linearisation need not include all components in an erroneous instance. We will assume that a linearisation of an instance can be obtained by applying a *linearisation function* to a tuple H of header node names and an instance mapping. An *r-local linearisation function* $L(H, I) \rightarrow s$ produces linearisations satisfying

1. Completeness: The linearisation of a correct instance is a permutation of all the components of the instance.
2. Determinism: If two instances of a storage structure have linearisations whose first n components ($n \geq 0$) are the same, and the values of those components are the same in the two instances, then the two linearisations have the same next component (or both linearisations contain exactly n components).
3. Locality constraint: There is an integer k such that, in an incorrect instance, either
   a) all erroneous components of the linearisation are in the last k components,
   or b) there is a subsequence of length k containing more than r erroneous components.

The first requirement simply means that, for a correct instance, all components are included in the sequence, and no repetitions are allowed. For an incorrect instance, neither of these conditions is required. The second requirement corresponds to the notion that the $(n+1)$'th component of the linearisation depends only on the first n components. Thus, the first component of a linearisation must be a header component; if two instances have the same value for that header component, they must have the same second component in the linearisation, and so on. The third requirement means that the linearisation function must terminate the sequence shortly after an erroneous component is included, unless there are "too many" errors close together. The occurrence of case 3(b) will be referred to as a *violation of the locality constraint.*

While our formal definition of an r-local linearisation function does not allow the function to return an indication of the correctness of the instance, we will assume that this is available. If it is not, a detection procedure can always be invoked on the instance.

As is the case in coding theory and the global theory of storage structure robustness, the definition of "erroneous component" is subjective. There is some instance which should exist and any differences are considered to be errors. Thus, even a correct instance will contain erroneous components if some other correct instance should exist. Of course, in such cases, the erroneous components will cause 3(b) to occur, because the linearisation function will simply observe a correct instance.

We can easily derive the following two properties of an r-local linearisation function from the above definition.

4. Detection: The linearisation of an incorrect instance includes at least one erroneous component.

Proof: Consider the correct instance corresponding to a given incorrect instance. At least one component of the correct instance must have a different value in the incorrect instance. Select the first such component in the linearisation of the correct instance. By the determinism property, the linearisations must agree up to and including this component, and hence the linearisation of the incorrect instance includes an erroneous component. □

5. Reasonableness: Every non-header component (nn, cn), with node name nn and component name cn, appearing in a linearisation is preceded (not necessarily immediately) by an edge component whose value is nn.

Proof: Suppose to the contrary that for some storage structure and instance, an r-local linearisation function returns a linearisation containing a component, (nn, cn), of a node not previously named by an edge in the linearisation. We can construct another instance with the same headers and the same values for all components preceding (nn, cn) in the linearisation, but which does not contain node nn. Then (nn, cn) does not occur in the linearisation of this instance, but the linearisations match up to the component preceding (nn, cn), contradicting the determinism property. □

Each r-local linearisation function can thus be implemented by an algorithm which only examines components of nodes connected to the headers, that is, by what can be termed a *reasonable procedure* [13]. Furthermore, the linearisation contains all components which the r-local linearisation function used in making its decision.

A storage structure is *r-local-detectable* if it has an r-local linearisation function. For example, the correction algorithms discussed above for 3-spiral lists and LB-trees are easily seen to incorporate local detection. The linearisation function is obtained by traversing the instance until an error is encountered, at which point a sequence is emitted containing all components for which the instance mapping has been interrogated. Although we will not provide formal proofs in this paper, double-linked lists are 1-local-detectable, 3-spiral lists are 5-

local-detectable, and LB-trees are 2-local-detectable. Neither CT-trees nor CTB-trees are locally detectable, as it is impossible in both cases to bound the length of the trailing subsequence containing an error.

An erroneous instance is *P-improvable,* (with respect to a linearisation function L) if procedure P can correct at least one of the errors in the linearisation produced by L. A storage structure is *r-local-correctable* if there is a procedure P, an r-local linearisation function L, and an integer k, such that every incorrect instance of the storage structure is P-improvable unless the linearisation produced by L violates the locality constraint (has a subsequence of length k containing more than r erroneous components).

It may seem odd that the definition appears to indicate a correspondence between r-correctability and r-detectability rather than between r-correctability and $2r$-detectability, as is the case for global robustness properties. This is because a "diagnosis phase" seems necessary, either as part of a $2r$-local detection procedure, or as part of the improvement procedure. The sufficient conditions for r-local correctability which we develop below require $2r$-local detectability, but we are currently unable to prove the necessity of this.

The definition of local correctability is phrased in terms of correcting at least one error under certain conditions. Once this error has been corrected, it is possible to apply the linearisation function to the modified instance. The resulting linearisation will have more correct components because of the determinism property. We can correct an error in this linearisation as well, and thus continue until all errors have been corrected or a violation of the locality constraint is encountered. Note that a given application of the improvement procedure does not necessarily correct the first error in the linearisation. However, because of determinism, if the first error is not corrected, it must appear in all subsequent linearisations, each of which contains more correct components following the first error. The improvement procedure cannot "ignore" such an error indefinitely, because an attempt to do so will eventually produce a linearisation containing only this one error, at which point it must be corrected.

Although direct application of the definitions requires a complete recreation of the linearisation after each error correction, in practice it is possible simply to discard the end of the erroneous linearisation and then extend it until another error is encountered (or the entire instance is processed). Thus, local correction will normally be performed in linear time, with each error adding a constant to execution time, rather than adding $O(\textit{size of instance})$ to execution time.

Clearly, the definition of local correctability given above is not constructive. It requires that a linearisation function and an improvement procedure be exhibited in order to show that a structure is locally correctable. The following sections describe practical techniques for constructing a linearisation function and an improvement procedure. Throughout, the fundamental idea of "voting" on the value of a component will be used.

5. Votes

We wish to derive sufficient conditions for local detectability and correctability based on a certain style of algorithm. Essentially, we wish to develop iterative algorithms for correction based on correcting one small "substructure" at a time. However, as we have argued in [11], we also have a clear need to avoid a combinatorial explosion if we wish to implement procedures for correcting multiple local errors. For this reason, we make use of the notion of a "vote" to structure our correction algorithms.

The development which follows will result in a class of algorithms which perform correction by correcting substructure instances which only depend on components already assumed to be correct. This approach seems intuitive in view of the reasonableness of a linearisation function. The substructure instances will be composed of components evaluated by votes, and will overlap each other in an instance. Each substructure instance will contain a small set of target components upon which rather stringent requirements will be placed, as well as diagnostic components which are much less constrained. The resulting sufficient conditions for local detectability and correctability of a storage structure seem to provide a natural characterisation of the intuition behind the two terms.

Consider a local detection algorithm which has placed some collection of components in the linearisation, believing some of them to be correct, and has not yet detected an error. Those components which the detection algorithm has checked sufficiently to believe them to be correct will be called *trusted*. Assume also that the algorithm has reached a "stable" state, in the sense that there are no other components whose values can be deduced from trusted components. Clearly, there must be fewer than k untrusted components in the linearisation at this point.

The detection algorithm must now begin a new detection step by selecting a component, c, to check next. Presumably, if there are components which have been used in detection but are not trusted, one of those will be selected. Because the algorithm cannot predict the value of c, it must expand the set of untrusted components to include at least r other components in order to guarantee the detection of r or fewer errors. In the worst case, r errors might have been inserted to c and $r-1$ other untrusted components, all agreeing on the incorrect value of c.

We will call c a *principal component;* the *target* associated with c will be the set of components whose values can be calculated using only the values of c and trusted components already in the linearisation.

In a 3-spiral, a principal component is a back pointer and the target components are the identifier and two forward pointers in the node pointed to. In an LB-tree, either of the two pointers to a node may be selected as the principal component. The target consists of the other pointer to the same node and the identifier component in that node.

As indicated at the beginning of the section, we wish to describe this detection activity in terms of voting. A vote will examine trusted components and untrusted components other than c itself in reaching a decision about the value of c. A vote is a predicate of three arguments: a tuple of header node names, a component, and a component test value. We will say that a predicate $V(H, c, x)$ is a *vote on c* if, assuming there are no erroneous trusted components,

1. V does not evaluate $I(c)$,
2. $V(H,c,I(c))$ *true* implies zero or multiple errors in c and untrusted components, and
3. $V(H,c,I(c))$ *false* implies one or more errors in c and untrusted components.

A vote thus performs 1-detection on a set of components, assuming the correctness of trusted components. If $V(H, c, x)$ is *true* we will say that V is a vote *for* x (as the value of c). Note that the Valid State Hypothesis is implicit in the 1-detectability of a vote: violations of it can have the same effects on votes as do multiple errors.

Since we assume each target component has a value which can be computed from c and other trusted components in the linearisation, a test can be associated with each target component, of the form $g(H, c, x)=u$, where u is the target component value and $g(H, c, x)$ is a function which generates u given that the value of c is x. If c is correct, an error in the target component will always be detected, but if c is incorrect, the test may be satisfied even if the target component is correct. In many cases, this test will also be a vote with the target component as the only untrusted component in it. In our examples, all pointers in targets have this property.

In the following, we will distinguish two types of votes: constructive votes and diagnostic votes. A vote $V(H, c, x)$ will be called *constructive* if there is a function, f, of trusted and untrusted components such that V is equivalent to evaluating the predicate

$$f(H,c)=x.$$

For convenience, we will extend the definition of constructive vote to allow $I(c)$ as a constructive vote. (Note that this trivial constructive vote does not correspond to any vote. The corresponding vote would be $I(c)=I(c)$, which detects nothing.) Any vote which is not a constructive vote will be called a *diagnostic vote.*

To perform r-local detection we not only need r votes on the principal component; we also require that the votes be sufficiently independent that no single error can adversely affect more than one of them. The concept we wish to define will be called distinctness of votes, but we first define distinctness of applications of votes. Consider two votes, V_1 and V_2, applied to the same component, c, and test value, x. The applications $V_1(H,c,x)$ and $V_2(H,c,x)$ will be called distinct if the untrusted components of the two votes do not intersect or at least one of the votes evaluates *false*. Two votes, V_1 and V_2, will be called *distinct* if, for all instances, components, and test values, the applications of the votes are distinct. Thus we may show that votes are distinct by showing that their untrusted components never intersect, or by showing that whenever an intersection does occur, at least one vote evaluates *false*.

Since in a correct instance all applications of votes must yield *true*, the untrusted components for distinct votes do not intersect in a correct instance. Superficially, it might appear that if two votes are non-intersecting in a correct instance, they must be distinct. However, as the set of components evaluated by a vote may change due to the presence of an error, this is not the case. Clearly, it is necessary but not sufficient for distinctness that votes be non-intersecting in a correct instance.

As an example, we may consider the distinctness of votes used when performing 4-local detection in a 3-spiral. Referring to Figure 2, the four votes on the principal component c are the following:

$$V_1(H,c,X) \equiv X.f_1 = A$$
$$V_2(H,c,X) \equiv X.f_2 = B$$
$$V_3(H,c,X) \equiv A.b.f_2 = X$$
$$V_4(H,c,X) \equiv B.b.f_1 = X$$

($Y.p$ denotes component p of the node named Y; b is the back pointer, and f_1 and f_2 are the two forward pointers.) V_3 and V_4 use untrusted back pointers, but these pointers and c are necessarily in three different trusted nodes. Thus, no vote evaluates c, and the only possible intersections of untrusted components involve the f_1 pointers evaluated by V_1 and V_4, or the f_2 pointers evaluated by V_2 and V_3. Considering the first case, the same f_1 pointer will be used by both votes only if c and $B.b$ have the same value, X. V_1 and V_4 both return *true* iff $A = X.f_1 = X$, implying $A.f_1 = A$. But $A.f_1$ is already trusted and known to have the value B. (Note that this reasoning can be considered to use an arbitrary assignment of components to trusted and untrusted sets. This assignment must be made in such a way that each component can be checked before it must be trusted. This issue is addressed by the theorem appearing in the next section.) Exactly the same reasoning may be applied to show that V_2 and V_3 are distinct. Thus, all votes are distinct.

6. Substructures

We define a *substructure of type t* to be a set of votes on a component with component name t. (The intuition behind the definition is that the components evaluated by the votes are a subset of the overall structure, but formally the substructure consists of votes not components.) If a substructure contains r distinct votes, it will be called an *r-detectable substructure.*

We define a *substructure instance* with principal component c to be the set of components evaluated by the votes in a substructure when the votes are applied to c. Clearly, c must be of the same type as the substructure. We divide the components in a substructure instance into three subsets:

1. Trusted components: the union of the trusted components of all votes.
2. Target components: as defined above, including c itself.
3. Diagnostic components: all non-target components in the union of untrusted components of all votes.

It is now possible to show that a storage structure is r-local-detectable if it can be appropriately covered by substructure instances, as stated in the following theorem.

Theorem: A storage structure is r-local-detectable if corresponding to every correct instance of the structure there is a sequence of r-detectable substructure instances satisfying

1. the targets of the substructure instances partition the instance, the size of all targets being bounded by a constant, m,
2. the trusted components of each substructure instance appear in targets of preceding substructure instances, and
3. all diagnostic components of each substructure instance appear in targets no later than the j'th succeeding instance in the sequence, for some constant j.

Proof: We must show how to construct an r-local linearisation function given the assumptions. It is claimed that the following is such a function: simply take each substructure instance in order, adding to the linearisation any of its target and diagnostic components which do not yet appear in the linearisation; for each substructure instance, evaluate all votes and test any target components which do not have a corresponding vote; if an error is detected, stop.

This satisfies the completeness property of the definition of an r-local linearisation since the targets partition the instance. Provided a deterministic ordering is chosen for the target and diagnostic components of each substructure, determinism will also be satisfied, since all decisions are based on components already placed in the linearisation. (Actually, it is possible that two or more sequences might satisfy requirements (2) and (3), but this can be resolved by picking a particular sequence according to some deterministic algorithm for selection of a "next" substructure instance.)

Finally, we show that the locality constraint is satisfied with $k=m*(j+1)$. This proof is given in three stages. First, we show that any error in a target will be detected if the substructure instance contains no more than r errors. Second, we show that if no errors are detected, all errors are in the last $m*j$ components of the linearisation. Third, we show that an erroneous instance terminates the linearisation with all errors in the last $m*(j+1)$ components. For the second and third stages, we assume that no subsequence of length k contains more than r errors.

We first show that any substructure instance containing at least one error in its target components and no more than r errors in all of its target and diagnostic components causes an error to be detected by the above algorithm. If the prinicpal component is incorrect, then a vote can evaluate *true* only if it contains another erroneous untrusted component. But for r distinct votes, this implies at least $r+1$ errors in total for all votes to return *true* erroneously. If the principal component is correct, any test on an incorrect target component will yield *false*, and at least one target component is assumed to be in error, so an error will be detected.

Now, consider the linearisation immediately after processing a substructure instance in which no error is detected. We show by induction that all errors are in the last $m*j$ components. This is clearly true for the initial, empty, linearisation, so assume it is true prior to the processing of a substructure instance in which no errors are detected. At most m components are added for this instance, hence all errors must now be in at most the last $k=m*(j+1)$

components. By assumption, there can be no more than r errors in this subsequence. Hence, the linearisation contains no more than r errors. Thus, each substructure instance in the linearisation contains no more than r errors. As shown above, an error will be detected if any target component is in error, hence all target components in all substructure instances are correct. By (3) in the statement of the theorem, all but the last j instances have all their diagnostic components in targets which are known to be correct, so errors can only exist in those diagnostic components of the last j instances which are in targets of instances not yet added to the linearisation. All of these are in the last $m*j$ components of the linearisation, as required.

When an error is detected, at most m components will have been added corresponding to the erroneous instance, and hence all errors will be in the last $m*(j+1)=k$ components. (Since the targets partition the instance, an error will be detected eventually. An error may be detected before the first erroneous target component is examined. The above reasoning is independent of whether the first erroneous component is a target or diagnostic component of the detectably erroneous instance.) □

This result and the discussion of distinctness above can be used to show that a 3-spiral is 4-local-detectable. The complete proof requires construction of special "initial" and "terminal" substructures, which we omit for brevity. In Section 7, such special substructures are constructed to demonstrate 2-local-correctability for 3-spirals.

Clearly, it is possible to perform local detection easily without explicit reference to the voting procedure described in this section, but, given the voting mechanism developed here, diagnosis and correction, which are very hard to perform on an *ad hoc* basis, are also quite easy.

7. Local Diagnosis and Correction

Once an error has been detected, it is necessary to determine which component(s) are in error and then correct them. In the case of local error correction, it is only necessary to locate and correct at least one erroneous component. It is possible to view the diagnosis phase as determining the correct value of the principal component, c. If the correct value is the current value of c, then other target components (as defined in Section 5) can be corrected. If the correct value is not the current value of c, then c can be corrected and possibly other target components as well.

If an r-local-correction procedure is using an r-local-detection procedure, then the diagnostic procedure will need to examine additional components in order to make this decision. (If a $2r$-local-detection procedure is used, it is possible that sufficient information is already available to make a diagnostic decision.) In any case, the diagnostic procedure must make use of at least $2r$ untrusted components since in the worst case, c and the first $r-1$ untrusted components examined might all be erroneous but in agreement with each other: the "weight" of these r erroneous components must be counterbalanced by $r+1$ correct components.

In order to perform diagnosis, as described above, within the framework of a "voting" algorithm, we must first generate a set of candidate values for c and then select the appropriate value by voting. Constructive votes will be used to generate the candidate values, and there must clearly be $r+1$ distinct constructive votes if at least one of them is to remain unaffected in the presence of r or fewer errors.

As we have indicated above, the votes used in selecting the correct value of c from the candidates provided by the constructive votes are called diagnostic votes. These diagnostic votes will be applied to each of the test values provided by the constructive votes. That is, if the diagnostic votes are $V_1,...,V_r$, and the distinct values returned by the constructive votes are $x_1,...,x_t$, we wish to evaluate

$$V_i(H,c,x_j) \quad (i=1,...,r;\ j=1,...,t).$$

To guarantee a correct decision after the constructive and diagnostic votes are applied, we require all of these constructive and diagnostic votes to be pairwise distinct. Note that this does not actually place any restriction between diagnostic votes used to test different candidate values, since distinctness concerns application of votes to the same test value.

Referring back to the votes described for a 3-spiral at the end of Section 5, the two non-trivial constructive votes for a back pointer, V_3 and V_4, are obtained by following the two alternative paths to the destination of the back pointer in the correct instance. The diagnostic votes available, V_1 and V_2, involve checking whether the forward pointers in the nodes reached by these paths match the last two nodes traversed. As an example, consider the 3-spiral in Figure 1 with two errors inserted. Assume the back(3) pointer in node 6 now points to node 2 and the forward(1) pointer in node 2 points to node 4. We then obtain one constructive vote for each of three nodes: the trivial constructive vote is for node 2; back(3), forward(1) from node 5 is a vote for node 4; back(3), forward(2) from node 4 is a vote for node 3. There is one diagnostic vote for node 2, the forward(1) pointer to node 4. There are no diagnostic votes for node 4. There are two diagnostic votes for node 3, since both forward pointers are correct. Thus, the error in node 2 has misled both a constructive and a diagnostic vote, but there are still more votes for node 3 than for either of the other nodes returned by the constructive votes.

For the LB-tree, if we select a sibling pointer as the principal component, the non-trivial constructive vote is the tree pointer to the same node. The diagnostic vote involves comparing the sibling of the node reached against the tree pointer which should point to the sibling.

Intuitively, the appropriate correction to a principal component, c, should be that value x_j for which there is the largest number of votes. Consider the hypothesis that x_j is the correct value of c. Let n_j be the number of constructive votes returning x_j, and m_j the number of diagnostic votes for x_j. Note that $n_1+ \cdots +n_t=r+1$. If x_j is the correct value, then $r+1-n_j$ errors must have affected other constructive votes, and $r-m_j$ errors must have affected diagnostic votes $V_i(H,c,x_j)$ $(i=1,...,r)$, for a total of $2r+1-n_j-m_j$. Clearly, this hypothesis is only tenable if the number of errors implied is less than or equal to r:

$2r+1-n_j-m_j<r+1$, or equivalently
$n_j+m_j>r$.

If this condition is satisfied, we call x_j an *appropriate correction* to c.

We are now in a position to prove the following:

Theorem: If a substructure instance with principal component c satisfies

1. the substructure instance contains distinct constructive and diagnostic votes as described above,
2. all trusted components are correct, and
3. there are r or fewer errors in the untrusted components,

then the original, correct value of c is the single value x_j such that $n_j+m_j>r$.

Proof: Consider the t distinct values $(t \leq r+1)$ returned by the constructive votes, and denote these by x_1 through x_t. As above, let n_i be the number of constructive votes returning x_i and m_i be the number of diagnostic votes $V(H,c,x_i)$ which return *true*. Assume, without loss of generality, that x_1 is the correct value for c. Then in the correct instance, we had $n_1=r+1$, $m_1=r$, hence $n_1+m_1=2r+1$, and $n_i+m_i=0$ for $i>1$.

Suppose s errors are in components used by the constructive votes in the correct instance. Then, because constructive votes are distinct, $n_1 \geq r+1-s$. The inequality occurs because a vote affected by multiple errors may fortuitously return the correct value. In the correct instance, the components evaluated by the constructive votes and the diagnostic votes on the (correct) value of c do not intersect, so, at most $r-s$ diagnostic votes on x_1 can be affected by errors, and potentially evaluate *false*. Thus

$$m_1 \geq r-(r-s)=s,$$

$$n_1+m_1 \geq (r+1-s)+s=r+1,$$

and x_1 is an appropriate correction. For $i>1$, the distinctness required between constructive and diagnostic votes gives $n_i+m_i \leq r$. (Since we can evaluate $V(H,c,f(H,c))$ for a constructive vote $V(H,c,x)$ and corresponding $f(H,c)$, this application must be distinct from the application of the diagnostic votes to the test value $x=f(H,c)$.) Thus, for any set of r or fewer errors affecting the votes, x_1 will be the unique appropriate correction. □

The complete diagnostic voting algorithm for a principal component consists of evaluating the constructive votes and counting them, evaluating all the diagnostic votes and counting them, and determining whether a single appropriate correction exists.

If the diagnostic voting yields a single appropriate correction, then the principal component can be set to that value (if its present value is different) and the values of target components can also be set to agree with the appropriate correction.

The definitions of substructure and substructure instance given in Section 6 do not depend on the use made of the votes, so we may define an *r-correctable substructure* to be a substructure satisfying the following

1. The substructure contains at least $2r$ votes and at least r of these are constructive. (Including the trivial constructive vote provides $r+1$ constructive votes in total.)
2. All of the votes are distinct.

For convenience, we will also say that a substructure is r-correctable (for any r) if it consists only of constructive votes with no untrusted components. Such substructures are used to set the principal component, and possibly other target components, to values which can be determined entirely from previous components. When a distinction is necessary, this will be referred to as a *degenerate substructure.* Such substructures could be merged with other substructures to meet the definition of an r-correctable substructure above, but this makes such substructures unnecessarily complex.

We are now able to prove that the voting algorithm actually can be used to perform local error correction.

Theorem: A storage structure is r-local-correctable if corresponding to every correct instance of the structure there is a sequence of r-correctable substructure instances satisfying

1. the targets of the substructure instances partition the instance, the size of all targets being bounded by a constant, m,
2. the trusted components of each substructure instance appear in targets of preceding substructure instances, and
3. all diagnostic components of each substructure instance appear in targets no later than the j'th succeeding instance in the sequence, for some constant j.

Note that this is exactly the theorem of Section 6, with "detectable" changed to "correctable."

Proof: We must show that an r-local linearisation function and an improvement procedure can be constructed given these assumptions. Since each r-correctable substructure is $2r$-detectable, and hence r-detectable, the existence of an r-local linearisation function follows from the theorem proved in Section 6. The improvement procedure simply performs the voting algorithm described and proven above and then makes a correction or corrections as described. This meets the conditions for an improvement procedure. □

In a 3-spiral, an instance of the general substructure consists of parts of six nodes, as shown in Figure 2. The trusted components are the back pointers giving the node names of the three nodes in the bottom row. (Actually, the vote must evaluate a whole sequence of back pointers, beginning at the headers, to locate these back pointers. All of these back pointers are also trusted components, but these "subordinate" trusted components will be ignored for simplicity in the example.) The principal component is the rightmost back pointer, marked c. The other target components are the identifier and the two forward pointers in the node marked X. The diagnostic components are the back pointers in nodes A and B, and one forward pointer in each of nodes Y and Z.

It is easy to verify that the substructure corresponding to this instance is 2-correctable, except for the distinctness of votes, however, the distinctness was demonstrated in Section 5.

There are also two degenerate substructures: an initial one and a terminal one. An instance of the initial substructure has a target containing the forward(1) pointers in H_1 and H_2, the forward(2) pointer in H_2, and the identifier fields in H_1 and H_2. An instance of the terminal substructure has a target containing the count, the back(3) pointers in the last two list nodes, and the forward(2) pointer in H_1. (H_0, H_1, H_2 denote the header nodes.) In a 3-spiral list containing n non-header nodes, the linearisation contains an instance of the initial substructure, followed by $n+1$ instances of the general substructure, followed by an instance of the terminal substructure, in order to cover all components with a target component in some substructure instance.

As a final remark, since an r-correctable substructure is $2r$-detectable, a storage structure which is r-local correctable by the method described above must be $2r$-local-detectable.

8. Discussion

The first part of this paper concluded with our formal definitions of local detectability and local correctability. If a storage structure is locally detectable, a detection procedure for the structure exists which is guaranteed to terminate in the vicinity of an error unless this vicinity contains more than r errors. To the extent that the traversal used by this detection routine matches that used by normal routines accessing the structure, it is possible to incorporate detection into "normal" processing. This is clearly easy for linked lists, but is not as clear for LB-trees. However, there exists a different 1-local-linearisation function for them which could be incorporated into an algorithm which proceeded from the root to a leaf node. This routine evaluates a constructive vote for each pointer on the direct path to the leaf. Each vote is obtained by following the pointer immediately to the left of the tree pointer, and comparing the value of the sibling pointer in that node with the tree pointer. (Since the level headers may be considered to lie along the left edge of the B-tree, there is always a pointer "parallel to" and to the left of any tree pointer.)

In this paper, we have presented hardly more than a definition of local detectability and some examples of storage structures which are locally detectable. Much work remains to be done to develop a general theory of local detectability which uses global properties of the storage structure to determine bounds on local properties. The local detectability of a structure is bounded by its global detectability, but it is not clear whether any of the global robustness properties are of further use in determining the local detectability of a structure. For example, a 3-spiral has a global detectability of 6, but is only 5-local-detectable.

For local correctability, the situation is somewhat better, as there do exist some useful sufficient conditions for local correctability based on votes. The type of algorithm required for local correctability in which an improvement procedure is iteratively applied to partially corrected instances seems at first glance to be from a much more restricted class than the algorithms which perform global correction in a storage structure instance. This is related to the notion of distance between errors, and the problems which we have had with earlier definitions of local correctability which were contradictory because they were unable to model explicitly the way in which the instance changed as the local correction algorithm operated. However, local correction algorithms are inherently able to correct a number of errors which is ultimately dependent only upon the size of the instance, while all global algorithms known to us have a constant upper bound on the number of errors they can correct.

We feel that the substructure approach to local detectability and correctability is quite intuitive, in spite of the often formidable formalisms surrounding it. The target components in a substructure are quite tightly related to the principal component, while the loosely constrained diagnostic components in any one substructure instance are dealt with more completely in some other target. This overlapping property of substructure instances seems particularly useful for reducing the complexity of multiple error correction. In particular, the voting algorithm nicely avoids the combinatorial explosion of symptoms and possible diagnoses which otherwise occurs in performing diagnosis of multiple errors. The necessary votes seem to "occur" quite naturally, although proving distinctness of votes can be tricky; both LB-trees and r-spiral lists were designed before the notion of a vote became well-defined.

One of the interesting tradeoffs in local detection and correction is between the use of a "pessimistic" and an "optimistic" algorithm. A pessimistic algorithm always attempts to perform the greatest amount of detection or correction, at the expense of more complex code and perhaps slower execution. An optimistic algorithm, on the other hand, performs a minimal amount of extra checking at the risk of being misled into proceeding (in the case of detection), or actually making inappropriate corrections. As an example, we mentioned the 2-local detection routine for 3-spirals used by the correction algorithm, the 4-local detection routine which always evaluates the non-trivial constructive votes, and the 5-local detection routine which is known to exist.

Rather curiously, it appears that the 2-local detection routine for 3-spirals detects up to 5 local errors (in a larger locality). Thus, if used purely as detection routines, the 2-local and 5-local detection routines seem to be equivalent. If used in conjunction with the 2-local correction routine, there is a difference: delayed error detection by the 2-local detection routine may cause improper "corrections" to be made, in cases which would be diagnosed as uncorrectable with use of the 5-local detection routine. This apparent equivalence between optimistic and pessimistic local detection requires further study.

For the moment, we have no good way of measuring the effectiveness of local correctability. Obviously, the probability of a correction algorithm succeeding depends critically on the distribution of errors within the instance. Unfortunately, any analytical evaluation of this probability for a locally correctable structure is intractable under even the grossest of simplifying assumptions, to say nothing of attempting to model a "realistic" distribution of errors introduced by, say, a misbehaving program. However, some simulation attempts have been made, and experiments are still in progress to evaluate the "effective" limit of how many errors can "usually" be corrected in particular storage structures. In particular, the local correction routines for k-spirals and LB-trees have been implemented and extensively tested.

Our main purpose has been to suggest the use of locally correctable storage structures as a viable technique to aid in the construction of fault tolerant and reliable systems. As such, we give examples of storage structures which are simple enough to provide significant robustness at acceptable cost. Obviously, the two main candidates in this regard are the 1-local-correctable 2-spiral linear list, which we did not discuss in detail here as it has been described elsewhere [11], and the LB-tree. Their use involves little overhead relative to "standard" implementations of linked lists and B-trees; their robustness can be exploited to increase overall system reliability significantly.

References

1. R. Bayer and C. McCreight, "Organisation and maintenance of large ordered indexes," *Acta Informatica* **1**(3) pp. 173-189 (1972).
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, "A robust B-tree implementation," *Proceedings of the Fifth International Conference on Software Engineering,* pp. 63-70 (9-12 March 1981).
5. L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," *Digest of Papers: Eighth Annual International Symposium on Fault-Tolerant Computing,* pp. 3-9 (June 1978).
6. K. P. Eswaran *et al*, "The Notion of consistency and predicate locks in a data base system," *Communications of the ACM* **19**(11) pp. 624-633 (November 1976).
7. J. J. Horning *et al*, "A program structure for error detection and recovery," pp. 171-187 in *Lecture Notes in Computer Science,* ed. E. Gelenbe and C. Kaiser, Springer Verlag, Berlin (1974).
8. D. B. Lomet, "Process structuring, synchronisation and recovery using atomic actions," *SIGPLAN Notices* **12**(3) pp. 128-137 (March 1977).
9. W. W. Peterson and E. J. Weldon Jr., *Error-Correcting Codes,* MIT Press, (1972).
10. D. J. Taylor and J. P. Black, "A locally correctable B-tree implementation," CS-84-51, Dept. of Computer Science, University of Waterloo (December 1984). Accepted for publication in *Computer Journal.*
11. 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).

12. 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).

13. 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).

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

15. J. H. Wensley *et al*, "SIFT: Design and analysis of a fault-tolerant computer for aircraft control," *Proceedings of the IEEE* **66**(10) pp. 1240-1255 (October 1978).