UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT

Guidelines for Storage Structure Error Correction

D.J. Taylor
J.P. Black

Data Structuring Group
CS-84-53

December, 1984

Guidelines for Storage Structure Error Correction

David J. Taylor

James P. Black

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

ABSTRACT

When using robust storage structures, writing correction routines is typically much harder than writing detection routines. Experience in developing correction routines has yielded a number of correction principles, which were reported previously. We have now had significant experience in application of those principles. In this paper, we present a re-evaluation of those principles, influenced by our experience in using them, and briefly describe some of the correction problems which have yielded to the principles.

1. Introduction

The use of robust storage structures provides the ability for a fault tolerant system to detect and possibly correct errors which affect the stored representation of data structures. A storage structure is j-detectable if any set of j errors applied to a correct instance leaves the structure detectably in error; the detection is made by a detection routine which returns a binary value indicating whether the instance is correct. Similarly, a storage structure is r-correctable if there exists a correction routine which can reproduce the original, correct instance, given only an incorrect instance assumed to contain at most r errors. A sufficient condition for a storage structure to be r-correctable is that it be $2r$-detectable and have $r+1$ edge-disjoint paths to each node in a correct instance.

Generally, writing detection routines is not difficult. If the structure is sufficiently well understood for update and access routines to be written, then a detection routine can likely also be written, with about the same effort. The situation for correction routines is quite different. Working directly from first principles, the implementation of a correction routine in many cases will seem a nearly impossible task. Generally, the development of a correction routine for any structure more complex than a double-linked list is a potentially very difficult problem. Thus, it is important to have a set of guidelines or principles which can assist in the structuring of correction routines, in order to make implementation easier. Indeed, without appropriate correction principles, implementation might effectively be impossible.

In addition, the correction principles provide some indication of appropriate ways to design robust storage structures. For a badly designed structure, correction may indeed be very difficult, but often a minor change to the structure will make correction easy.

This paper does not repeat the extensive set of examples which were previously used to justify the correction principles [3]. Rather, in the next section, we expound upon the principles and explain their usefulness generally. Some of the principles no longer seem as useful as they did originally; in those cases we explain why this is so. In Section 3, we describe how the principles have been applied to solve a number of difficult correction problems. Finally, in Section 4 some general conclusions are presented.

2. Correction Principles

The following principles are presented in an order somewhat related to their importance. In particular, we now feel that the first three are much more important than the last two.

1. Prevent loops and unbounded foreign traversal.

   Clearly, a correction routine must not be allowed to go into an infinite loop or read an arbitrary number of nodes which do not belong to the instance being corrected. Since any stored count field could be in error, it cannot be used to provide an upper limit on number of nodes visited, as might be appropriate in other routines. Thus, it is important that any correction routine make some checks inside any loop which traverses the structure being corrected, so that a potential infinite loop in the correction routine will be transformed (at least) into a detected error.

2. Partition the structure, preferably into determining sets

   The importance of partitioning the structure was recognised very early in our experience with correction routines. More recently a theorem, which seems obvious in retrospect, has highlighted the importance of partitioning into determining sets. (A *determining set* is a subset of the structural data which allows reconstruction of all other structural data. For a more complete discussion, see [4] or [1].) The theorem simply states that if a storage structure is $2n$-detectable, contains $n+1$ disjoint determining sets, and the algorithm for reconstructing all structural data from each of those sets is linear time, then n-correction can also be performed in linear time. Thus, if a partition into a sufficient number of determining sets is available, then a "cookbook" method exists for producing an efficient correction routine. In general, partitioning into independent sets of data is useful, even if these are not determining sets, as illustrated by an example in the next section.

3. If you can't decide, guess.

   This has turned out to be a very important principle. Very often it is much too difficult to decide exactly which field is in error, but it is possible to select a small set either of fields which might be in error, or of alternatives to be explored. By guessing, and using an error detection routine to validate the guesses, a simple correction routine can often be constructed. Although the guessing might seem to be inefficient, if the number of guesses is bounded by a constant, and the detection procedure is linear time, then the correction routine will also be linear. In particular, the determining set correction procedure mentioned in the preceding paragraph relies on guessing, but still achieves execution time linear in the size of the instance being corrected.

4. A coroutine structure may be useful.

   The idea here is that two or more disjoint sets of structural data may be traversed in parallel, checking for discrepancies. While this may be useful in some cases, it now appears that guessing which set is correct generally yields a simpler correction routine.

5. Use a fault dictionary.

   Here, if a complex set of conditions is observed, an appropriate tabular structure may be used to pick a routine suited to dealing with the situation. This may easily arise when using a coroutine structure, since each coroutine will provide a status value, and agreement/disagreement between the coroutines provides an additional piece of status information. If such a situation must occur in the correction routine, then a fault dictionary is indeed an appropriate tool. However, experience now indicates that often other approaches, notably the use of guessing, can be used to avoid such complexities, and hence avoid the use of fault dictionaries.

We conclude this section by giving the determining set correction method mentioned above. The procedure is so simple, that it is somewhat surprising it was not discovered earlier. The reason probably is that it makes crucial use of guessing, and for some time, guessing did not seem a "respectable" correction technique.

The algorithm simply guesses in turn that each of the determining sets is correct, and then investigates the consequences of the guess. Having guessed that a particular determining set is correct, it uses that set to reconstruct the entire storage structure instance. If the reconstruction requires the changing of too many fields (more than the correctability of the structure), the guess is rejected. If the reconstructed structure is not accepted by the detection routine, the guess is also rejected. If the detection routine accepts the structure, correction is complete. If a guess is rejected, any changes made during the processing of that guess are undone before proceeding to the next guess.

If the $(n+1)$-determined structure is assumed to contain at most n errors, then at least one determining set must be free of errors, and so the desired correct instance will eventually be reconstructed. Since the desired instance is the only correct one reachable within n changes (because of the $2n$-detectability), the algorithm properly ensures that no more distant correct instance is "accidentally" created.

3. Application of the Principles

In this section, we describe how application of the first three principles has greatly assisted in the construction of a number of correction routines. A particularly simple example is that direct application of the determining set correction procedure described in the last section has produced an extremely general correction procedure for linked lists which supersedes several previous correction procedures. Since, for a linked list, the determining set reconstructions are very simple and regular, a single correction routine can easily adapt to an arbitrary pointer structure. Previous correction routines, such as the k-linked list correction routine [3], may achieve slightly better efficiency on specific list

structures, but we intend to measure performance of actual implementations before drawing any definite conclusions about relative efficiency.

The previous paper describing correction principles mentioned a very difficult problem: single error correction in the chained and threaded binary tree. (A chained and threaded binary tree has standard thread pointers in logically null right pointer fields, and uses logically null left pointers to form a single-linked list, the chain, of all nodes with no left subtree.) At the time that paper was written, no correction technique was known with better execution time than $O(n^3)$ for an instance of n nodes. This technique was too grossly inefficient to be implemented, so there was effectively no way, at that time, to correct chained and threaded binary trees. A correction routine has now been developed and implemented which depends heavily on the guessing and partitioning principles.

In order to achieve proper partitioning, it was necessary to modify the storage structure slightly. In the original version, there was a tag bit attached to each pointer indicating the type of that pointer (tree pointer or chain for left pointers, tree pointer or thread for right pointers). The problem with this encoding is that a single error can change both the value and the type of a pointer. Instead of tagging the pointer with its own type, a better arrangement is to tag a pointer with the type of its target node. That is, each pointer now carries two tags, indicating what types of pointers are stored in the node to which it points. The result is a stabler partitioning of pointers into two sets: the chain and thread set, and the tree set.

In this partition, only one of the sets (the tree pointers) is a determining set, but it is still possible to achieve correction. The procedure uses guessing at two levels. First, it guesses which set contains no errors. When guessing that the tree pointers are correct, the actions to be taken are straightforward, since the set is a determining set. When guessing that the chains and threads are correct, the situation is more difficult. However, the chains and threads do allow all the nodes in the tree to be located, and in a correct tree there should be exactly one tree pointer to each node. By counting the number of tree pointers to each node, we eventually find either (1) one node with no pointers to it and one pointer to something which is not a tree node, or (2) one node with two pointers to it and one with no pointers. In the first case, correction is straightforward. In the second case, we apply guessing once again, to decide which of the duplicate pointer values should really be a pointer to the node with no pointers. Of course, some ingenuity is required to prevent this algorithm from taking too much time or space, but this can indeed be done [2].

Although the correction routine described in the previous paragraph is satisfactory in many respects, its pointer-counting technique does not seem entirely satisfactory from an intuitive viewpoint. Also, its execution time is $O(nh)$ for an n-node tree of height h, because it maintains a list of nodes to which only one pointer has yet been encountered. The maximum number of nodes in this list is approximately the height of the tree, thus repeated searches of the list prevent the correction from taking place in linear time. A clever organisation for the list could decrease the execution time to $O(n \log h)$, but could not make the execution be linear time. If the structure could be partitioned into two determining sets, then a linear-time, and more appealing, correction routine

would become available. It is indeed possible to do this without significantly increasing the complexity of update routines. The necessary additional data is simply a tag in each node indicating whether the node is a left son or a right son. The chain and thread pointers by themselves do not form a determining set, because in general, many binary trees have identical chain and thread structure. However, knowing whether a given node is a left or right son in the tree provides sufficient additional information to enable a unique reconstruction.

Another correction problem which had been outstanding for a long time involved compound storage structures [5]. (A compound storage structure is one in which several storage structures share exactly the same set of non-header nodes, the data in each node being partitioned between the structures.) The detectability of a compound structure is often greater than that of the component structures. Coupled with the attendant increase in edge-disjoint paths to a node, this in turn may increase the theoretical correctability of the compound beyond the correctability of the component structures, or even beyond the sum of their correctabilities. It was easy to calculate the correctability of a compound storage structure, but there was no known efficient method for achieving that correctability when it was greater than the minimum of the correctabilities of the component structures.

In some sense, the top level of such a correction routine is obvious: the routine must guess which component of the compound to use as a basis for correcting the others. In this case, we do not guess that one is correct, but that it is sufficiently close to being correct that its own correction routine will succeed. If the correction routine succeeds, then we know the exact set of nodes which must be in each other component, but the remaining problem is how to make effective use of this information when correcting those other components. There is apparently no completely general algorithm, but we have now solved the problem for a number of special cases. Here we present only the simplest case, which occurs when the component is a linked list.

For regular linked lists, the correctability of a compound structure is strictly less than $2m$, where m is the minimum of the number of pointers in each list being combined. Thus, at least one of the m determining sets in each list must contain zero or one errors. We guess that this is true of each set. For each set, we determine the number of pointers, in that set, to each node. The situation is essentially the same as in the chained and threaded binary tree correction algorithm described above. We find a discrepancy in the number of pointers to certain nodes, and if necessary apply guessing again to discover the necessary correction. Having corrected the set, we can use it to reconstruct the rest of this component, and continue correcting individual components of the compound structure in this fashion until the entire structure has been corrected. Note that this algorithm actually applies guessing at three levels: which component of the compound can be corrected independently, which set of pointers in each component contains fewer than two errors, and which duplicated pointer value to change. In spite of all this guessing, the algorithm has execution time linear in the number of nodes in the instance.

Two examples in this section rely on pointer counting for part of the correction algorithm. It does not seem appropriate to make pointer counting a sixth correction principle, but it is a useful technique in some circumstances. In particular, when a subset of structural data is not a determining set but does contain exactly one pointer to each node, pointer counting may be the most effective correction technique.

4. Conclusions

In this paper, we have attempted to present a useful summary of principles for constructing correction routines. We have found these principles to be useful in solving a number of correction problems which previously seemed intractable. In many cases, the correction routines described above have been implemented. Generally, we have found that routines designed according to these principles are well-structured and (relatively) easy to understand. We intend to continue applying these principles to additional correction problems. We also intend to implement the, as yet, unimplemented correction routines, in order to gain further practical experience, and observe the behaviour of such routines in practice. Previous experience indicates that a correction algorithm which appears simple in concept can sometimes encounter enormous practical difficulties. Thus, while we believe that the binary tree correction based on determining sets and the correction of compounds of linked lists will be easy to implement, we would like to determine exactly how easy implementation is.

References

1. J. P. Black and D. J. Taylor, "A model for storage structures, encoding, and robustness," CS-84-45, Dept. of Computer Science, University of Waterloo (December 1984).
2. J. L. Shepherd, *A robust chained and threaded binary tree implementation,* M.Math. Essay, University of Waterloo, Ontario, Canada (October 1983).
3. 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).
4. 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).
5. 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).