A CASE STUDY
IN FAULT TOLERANT SOFTWARE

J. P. Black, D. J. Taylor,
and D. E. Morgan

Department of Computer Science
and
Computer Communication Networks Group
University of Waterloo

May, 1980

Research Report CS-80-24

Abstract

The addition of redundancy to data structures can be used to improve a software system's ability to detect and correct errors, and to continue to operate according to its specifications. A case study is presented which indicates how such redundancy can be deployed and exploited at reasonable cost to improve software fault tolerance. Experimental results are reported for the small data base system considered.

Key Words and Phrases: software reliability, software fault tolerance, robust data structures, redundancy, error detection, error correction, audit programs.

1. INTRODUCTION

Reliability is becoming an increasingly important aspect of large software systems, due to their constantly increasing complexity, and the growing cost of system failure. One way of increasing software reliability is to give the system the ability to detect and correct errors, thus enabling it to continue to operate according to its specifications. Our purpose in this paper is to illustrate ways of applying redundancy to storage structures to improve fault tolerance, including the application of some fairly general theoretical results to a specific example.

The remainder of this section presents our terminology, and then provides an overview of our approach to the case study. A sample data base system, which was our point of departure, is described in Section 2. Section 3 describes the redundancy which was added to make the system more robust, and Section 4 relates how such redundancy can be exploited. Finally, Section 5 describes the experiments performed to verify the effectiveness of the modifications, and Section 6 presents our conclusions, a summary, and suggestions for further work.

Using Randell's definitions [2], reliability measures the degree to which a system adheres to a presumably authoritative specification of its behaviour. Deviation of the system from its specifications is a failure, which is due to the presence of a fault. An error is that part of the overall system state which is incorrect: it results from the fault. Thus, in a data base system, repair of a fault may involve changing or replacing a program, while correcting an error involves changing the contents of the data base.

As defined by Avizienis [3], there are two complementary approaches to improving software system reliability: fault intolerance or fault avoidance, and fault tolerance. Fault intolerance attempts to ensure the system is reliable before it is put into production, by such techniques as structured programming, design, and testing, as well as more formal techniques such as proofs of correctness. These techniques may be complemented by the fault tolerance approach, whose objective is to prevent faults (of whatever origin) from producing failures during normal system operation. Examples of fault tolerance techniques are recovery blocks [2], process checkpointing, rollback and recovery, and the use of redundant information to detect and correct errors.

The following definitions will be used in discussing data structures. A data structure is defined to be a logical organisation of data. A storage structure is a representation of a data structure. The representation specifies whether nodes are to be adjacent or connected by pointers, what pointers are used, and so on. An encoding of a storage structure is its representation on a particular storage medium. The encoding specifies how pointers are represented (absolute, relative, etc.), what fields are packed into a single word, and so on. Thus, "binary tree" is a data structure; a representation in which there are pointers from each node to the left and right sons of the node is a storage structure for a binary tree; and if we also specify that pointers are stored as absolute addresses, that is an encoding of a binary tree. (This terminology is adapted from Tompa [8].)

Our research is concerned with the addition of redundant structural information to storage structures. The forms of structural redundancy we consider are: counts, identifier fields, and extra pointers. (These are described in Section 4.) We have previously developed some general theoretical results [5, 7]. Our purpose here is to describe an empirical investigation with a particular set of data structures which are sufficiently complex that our current theoretical results are not able to deal with them.

We began with a small data base system, the Example System (EXSYS), which contained staff data for the University of Waterloo, and was developed during a course in data base management systems. We assumed that the data base, when subjected to a source of errors, would require some form of redundancy and supporting software to detect and correct the errors. We made no assumptions about their source: the errors could be due to hardware faults, software faults, user mistakes, intelligent adversaries, or unrelated system crashes.

As mentioned above, our basic technique was to add redundant pointer fields, count fields, and identifier fields to EXSYS' data structures so that errors could be detected and corrected. In doing this, we both influenced and profited from the theoretical results, which were developed more or less concurrently. In order to make use of the redundant structural information, we made extensive use of an inter-related set of error detection and correction routines, which we called audits. (The term "audit" in this sense was originally used by Bell Laboratories [1, 4].)

Our experimental method involved the use of a "mangler" to pseudo-randomly inject errors into the data structures as they were written to external storage. This permitted us to draw conclusions about the effectiveness of the audits and the added structural redundancy.

2. THE EXAMPLE DATA BASE

As seen by the user, EXSYS provides the ability to query and update a data base of university staff and faculty information. Each staff record contains about one hundred characters of information, and may individually be added, deleted, and changed. The query facility is based on the dynamic creation and use of attribute lists, which are lists of staff keys for those records having a particular value in a given field. Examples are the "science" list of all staff members whose faculty code has the particular value which indicates "science", and the "dean" list constructed on the academic title code field. There are commands to create and name an attribute list, delete an attribute list, find all staff members on each of a set of lists, and to print all attribute list names for a staff record field. We were not particularly interested in the set of commands available, but rather in the data structures which supported them.

EXSYS uses two files: the staff file and the index file. Access to the staff file by primary key (staff number) is by an externally-chained hash table residing in the index file. In addition to non-structural information, each staff record in the staff file contains a field which links successive records on a given hash chain. The index file consists of special records (called the master table, master list, and hash table), free index file records, and linked chains of attribute list blocks. Each block contains some number of staff file keys which point to some of the staff records on the attribute list. The master list contains the headers for the attribute lists, including their user-defined names, and the value being indexed for each list. There are two distinguished attribute lists, called the "$staff" and "$free" lists, which group all staff records in the staff file according to whether they are allocated ("$staff") or logically empty ("$free").

Record Ø in the index file always contains the master table, which is EXSYS' most important record. It contains pointers to the master list and hash table records, and to the linked list of free index file records. It includes a description of the staff record fields, and for each field, pointers to the first and last attribute list headers for the field. (As mentioned, these headers reside in the master list.)

Figure 1 shows the original form of the EXSYS data structures. This is their unimproved form, which we call Version Ø. Version 1, which we will not discuss in detail, represented our initial attempt to add redundancy and to use it for error detection and correction. As a first attempt to integrate an audit system with a data base management system, Version 1 was moderately successful. Experimentation with Version 1, and the development of some theoretical results motivated us to further improve EXSYS. This resulted in Version 2, which retained the overall design of Version 1, and which we describe in the next section.

3. ADDING REDUNDANCY

As we have stated, we are interested only in the addition of redundant structural information to storage

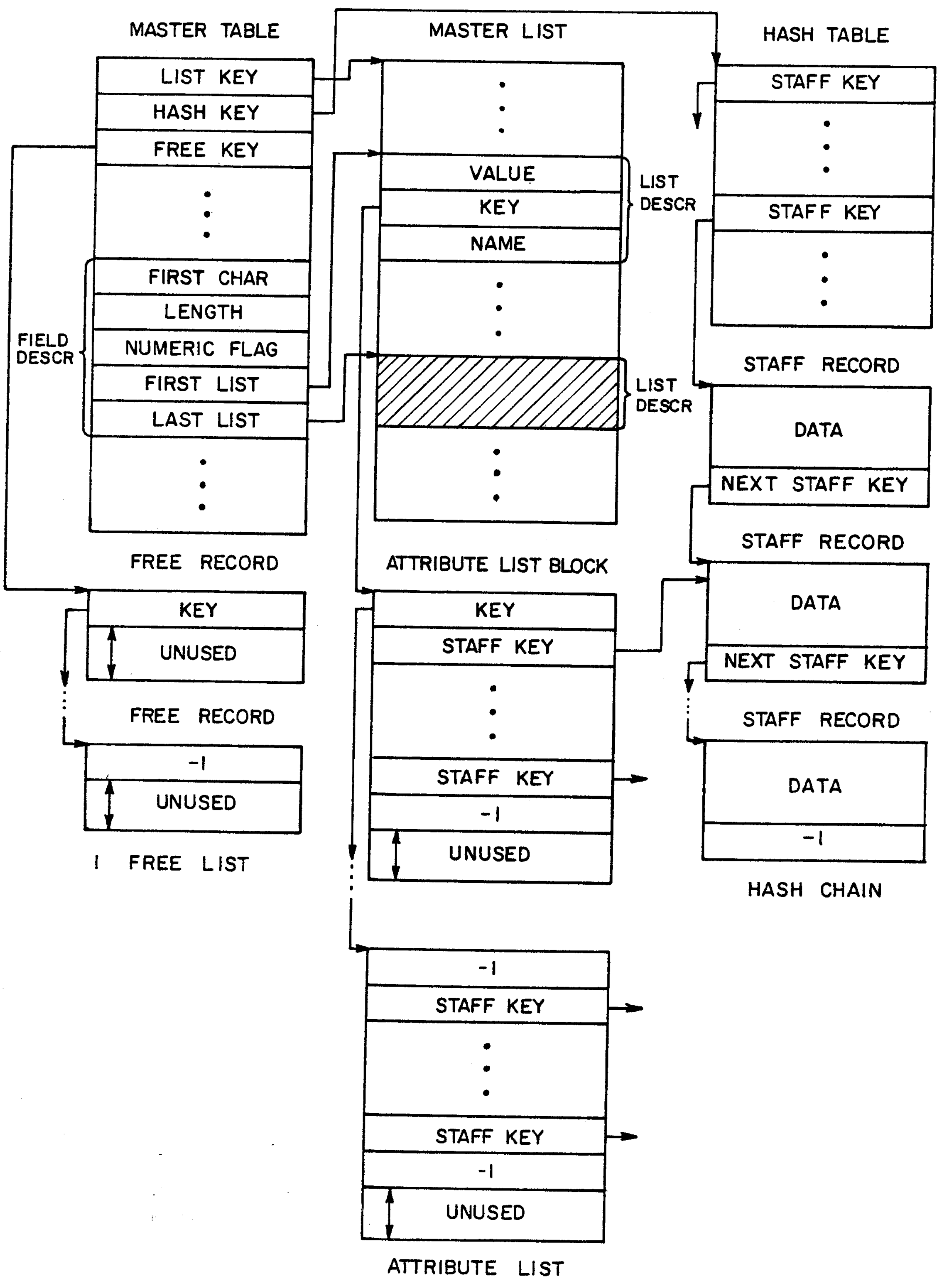

FIGURE 1: EXSYS Data Structures, Version 0

structures, as our theoretical results are not yet able to present any general, uniform approach encompassing both "data content" and structure. In particular, we add redundant pointer fields, identifier fields, and count fields. An identifier field contains a value which is unique to the type of node and the specific instance in which it occurs. A redundant identifier field is one whose value in a correct storage structure instance may be uniquely determined from pointer data alone. A count field contains a count of the number of nodes in an instance or in a defined subset of the instance. While it is not required by the theoretical results, it seems clear that these redundant fields should not simply be duplicates of the non-redundant fields, since in that case, some types of faults, such as software bugs, would almost certainly cause the same error to appear in all copies of a field. An example of a redundant set of pointers is the set of back pointers added to a single-linked linear list, forming a double-linked list.

While we have defined the types of redundancy which we use, we have yet to describe their usefulness. The error detection and correction properties of a storage structure are stated in terms of changes. A change is an elementary modification to an instance of a storage structure, where the definition of "elementary" may be adjusted to suit the application. Here, we will consider it to be any change to a machine word, but in other examples, a change could range from the modification of a single bit to that of a sector or track on a disk device.

Formally, we define an instance of a storage structure to be correct when a "detection procedure" applied to the instance returns "correct". If all sets of N or fewer changes applied to a correct instance yield an incorrect instance we say the storage structure is N-detectable. Similarly, a storage structure is N-correctable if there is a procedure which, for all sets of N or fewer changes, can take a correct instance modified by that number of changes, and recreate the correct instance.

As far as EXSYS is concerned, the best illustration of the use of the theoretical results is seen in the hash chains of Version 2. In Version Ø, the hash table contained an array of pointers to the start of the hash chains, and each chain was terminated by a null pointer. Obviously, such a structure is not very robust, as changing a single chain pointer to null truncates the chain, causing some number of staff records to become lost.

On the other hand, the theory shows that a double-linked list with an identifier field in each node and a node count is 2-detectable and 1-correctable. In Version 2, we thus added two words of structural information to each staff record: an identifier field whose value indicates "staff file" and contains the hash chain number, and the back pointers required to make the hash chains into double-linked lists. We also added a fourth special record to the index file: it contains the back pointers to the last staff record on each hash chain. For space reasons, we did not implement a count field for each chain, but rather a total count of records on all chains. Thus the individual hash chains are not 2-detectable, but the complete collection of hash chains is.

One somewhat subtle point arose in designing the robust hash chains. At first, we tried to use null pointers to terminate both the forward and backward chains, treating nulls as implicit pointers to the header (hash table) records. This did not work properly, and the theoretical results indicated that each chain should point to its own header, not a general header for all chains. Thus, chains are now terminated with special values which include the hash chain number, allowing the right entry to be located in the appropriate hash table. Once this point was cleared up, these modifications were easily exploited through the inclusion of the simple algorithm from [5] to perform 1-correction on each hash chain.

Intuitively, the 1-correctability results from the identifier fields, and the redundant back pointers. The identifier fields permit immediate identification of a pointer which has been changed to point outside of the instance. When this occurs, or when other pointer changes are detected by comparing forward and back pointers, correction can be achieved by traversing the list in the reverse direction, and taking a "vote" among those pointers involved. (For more details, see [5, 6].) As for the (intuitive) justification of the addition of count fields, the simplest example concerns a single-linked list with a count and identifier fields. The latter permit detection of changes which make a pointer point outside the list, while the count allows detection of changes which shorten or lengthen the apparent list.

Applying this type of reasoning to EXSYS, we added a variety of redundant information, mostly to the index file. This included fields containing the file size of each file, and a count of the number of allocated staff records, all in the master table. All records have identifier fields, and attribute list blocks have a secondary identifier field unique for each list. Each attribute list has block and key counts in its header, as well as the (redundant) number of the field on which the list is constructed. For each field, the master table entry for it contains a count of those staff records appearing on no attribute list for the field (the "none" count). Finally, free staff records have a unique identifier value in their identifier and pointer fields; this increases the number of changes required to make a free record appear allocated, or vice versa. A related design consideration was to make all the index file lists as similar as possible. Thus, the same set of low level routines may be used for the index file free list, the $staff and $free lists, and all attribute lists.

The purpose of our study was to investigate structural redundancy. In order to provide some redundancy for the non-structural information in the staff records, we added a checksum of those fields not "protected" by other redundancy.

Figure 2 shows EXSYS' data structures as they finally stabilised in Version 2. Redundant fields added to improve reliability are indicated by an asterisk (*).

4. EXPLOITING THE REDUNDANCY

The key to achieving fault tolerant software is redundancy, but redundancy itself is useless unless it can be exploited at reasonable cost, in terms of extra storage, CPU time, and I/O operations. The purpose of this section is to show one way of making use of redundant structural information, while the next section shows that the cost is not only reasonable, but that it may be easily adjusted.

Proper deployment of redundancy permits errors to be detected in a system's data structures. This detection may be "in-line", that is, achieved by code inserted in each program, or by audit programs invoked periodically or when trouble is suspected. The choice between the two detection methods is delicate, and we doubt that any hard and fast

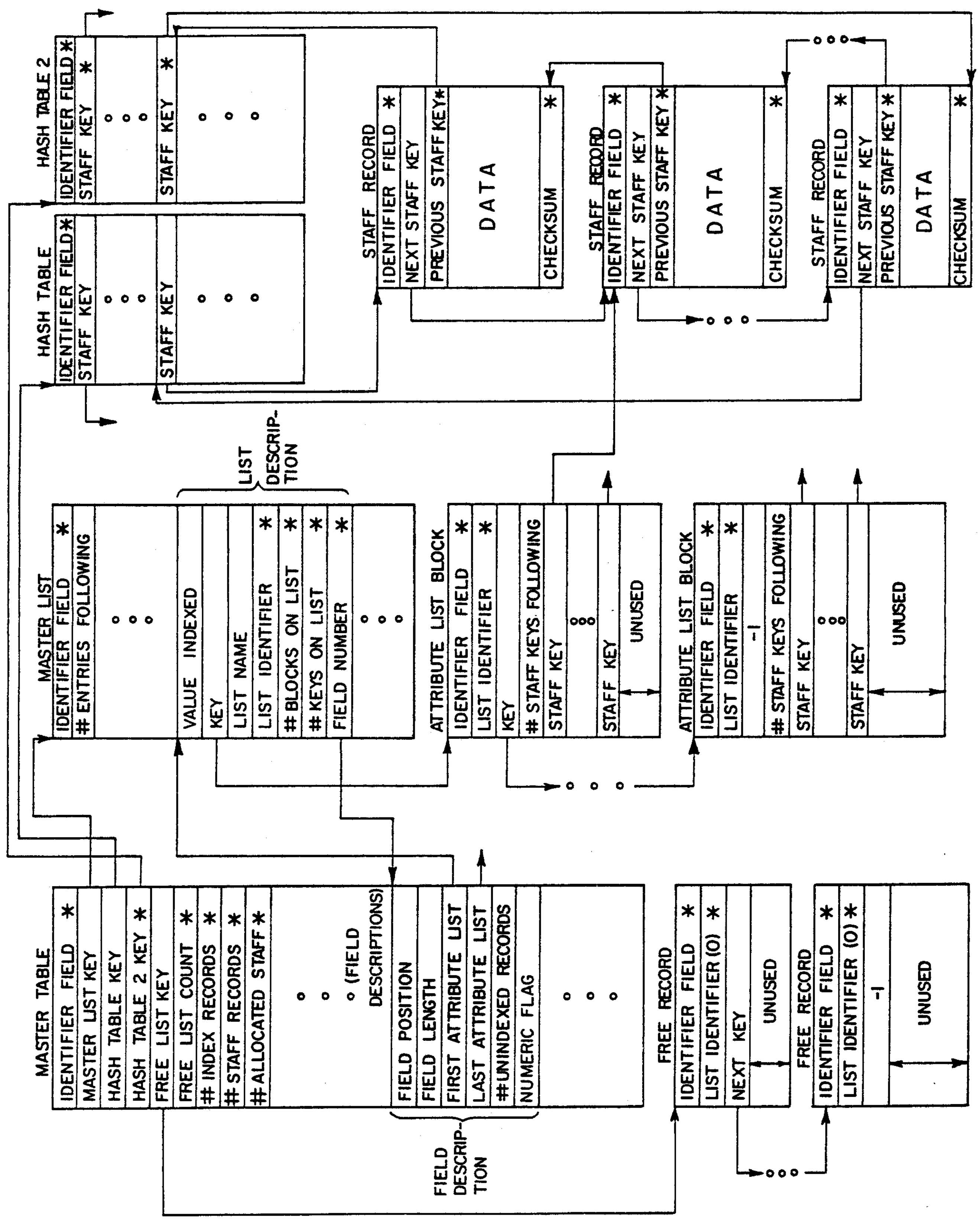

FIGURE 2: EXSYS Data Structures, Version 2

rules exist. The advantage of in-line checks is that they may detect errors sooner, and before they propagate. In some cases, as with double-linked lists, a detection procedure can easily be integrated into a list traversal routine, especially since this data structure is quite simple. On the other hand, merging detection with normal processing can easily introduce much undesirable complexity into a program, obscuring the "normal" logic flow. Furthermore, in-line checks introduce a constant, unavoidable overhead which may be unacceptable, since the data structures are expected to be correct most of the time.

Especially for a complex set of inter-related data structures such as those of EXSYS, using audit programs has the virtue of leaving the main system largely unobscured by detection or correction logic. As a side effect, changing or debugging the audits then has little effect on the correctness of the main system. However, the frequency of periodic audit invocation must be adjusted in view of the expected error rate, and the cost of audit program execution. For our experimentation with EXSYS, we chose a compromise solution which included some basic in-line checks, although most of the error detection and all of the error correction logic was included in the audit system.

Besides the normal checks performed on user input to EXSYS, other in-line checks were added which would be unnecessary in an error-free system. For example, the routine which searches an attribute list for a given staff file key also checks the keys on the list for ascending sequence. Other "easy" checks are made: for example, when a staff record is read because it is supposed to be on one or more attribute lists, it is verified that its fields' values are those specified by the attribute lists in question. If any of these in-line checks fails, an appropriate call is made to the audit system.

As we have mentioned, the audit system is invoked either periodically, or on an emergency basis when called from an in-line check. For periodic invocation, each user command causes a subset of the "audit threshold counters" to be incremented. If, at the end of a command, one or more counters have exceeded their thresholds, the counters are reset, and the audit system is invoked. By varying the thresholds, the relative frequency and hence the overhead cost of the audits may be adjusted as required.

The audit system is quite large. It is coded in approximately 40 pages of high level language code, compared to about 30 for the main system. (Common routines are counted twice.) The difference is somewhat exaggerated in that the audit system is more or less "complete", while the main system should be extended and improved to be used in a production environment; on the other hand, we do not claim that achieving high levels of reliability is easy or inexpensive.

Thus it might appear that the probability of a program error occurring in the audit system may be greater than the same probability for the main system. However, the detection routines are easily tested on correct instances, in which case no modifications are made by the audits. This leaves the possibility of a correction routine propagating or aggravating a previously detected error. This indeed occurred during system development, in many cases causing us to revise the audit system design rather than merely debugging one audit. On the other hand, we have observed during our experimentation that the audit system is fairly robust in a certain sense: while it often failed to perform the minimal amount of work required to produce a correct instance, it always managed to correct detected errors, and it never terminated itself because of apparent looping.

Entry to the audit system is made through the audit scheduler routine, whose parameters indicate the audit requested, a parameter to be passed to the audit, and a "re-dispatch" flag (explained below). The scheduler creates an audit queue entry from its parameters, and if no audit is currently being executed, proceeds to a loop which empties the queue by calling the audit at the head of the queue and removing the queue entry when the audit terminates. Each call causes an entry to be made in the scheduler's "scoreboard", which may be tested by an audit to determine if audits on which it depends have already been invoked.

When an error is detected, an audit may recursively call the audit scheduler to place another entry in the queue on a priority basis. (These recursive calls do not proceed past the point of adding an entry to the queue, since the queue was not empty when they began execution.) This rather complicated design reflects (in part) the complexity and underlying interdependencies of EXSYS' data structures. We suggest, however, that the mechanism is quite general, and is easily adapted to other data structures and audits.

Ideally, one would like to be able to audit each part of the data structure separately. If this were possible, any one audit could be called and executed without regard for the others. But clearly, it is not possible: how could one pretend to verify a particular attribute list without verifying that the master list containing the list's header is at least "probably" correct? Another argument against independent audits is that one audit may detect an error which it could most easily correct by calling another audit. Additionally, the first audit might wish to re-invoke itself after having called the second. This is implemented by the "re-dispatch" flag. Scheduling an audit with re-dispatch causes the caller's queue entry to be reinserted in the audit queue, according to its priority, when it terminates execution. The use of an audit scheduler encourages modular design of audits, and makes their addition or redesign simpler, as the audit scheduler is table driven. Audits may be made logically distinct and smaller, while their complicated interdependencies are realised through the combination of audit priority, the re-dispatch facility, and the scoreboard.

The audits themselves may be categorised in several ways: according to cost in terms of disk I/O, in terms of priority, or in terms of which part of the data structure concerns them.

The cheapest audits are the "quick" audit, "field" audit, and "none count" audit, which at one point were all one audit, and which (more or less) confine themselves to checking the core-resident tables. The quick audit is always run before any other; even calls to audits from within the audit system are preceded by calls to the quick audit. It is driven by a table with entries for the three special records pointed to by the master table (master list, forward hash chain pointers, back hash chain pointers). Each entry contains the identifier field value, key location in the master table, buffer address, and the address of the corresponding audit program.

At one point, the quick audit contained code to check the consistency of the field descriptions and list descriptions of the master table and master list, including the "none counts" of all fields. It became apparent to us that this was undesirable for two reasons. A certain set of interdependencies could cause a circularity in the call chain in some cases. Secondly, when any attribute lists are audited for a field, the none count for the field is not meaningful until all list audits have been completed. This implies the none counts should only be checked after all attribute list audits have completed. Thus the field audit was created with an appropriate intermediate priority, and the none count audit with the lowest priority. An external entry to the audit system generally results in three queue entries: the requested audit, the field audit, and the none count audit, with the quick audit being called implicitly before each.

The "high-level index" audit is slightly more costly, and has a high priority. It verifies the overall structure of the index file: the free list and all attribute lists are checked as much as possible without reference to the staff file. Errors detected may cause a "map" audit to be run to attempt to recover lost index file records, a full attribute list audit to be run, or a "$staff/$free" audit to be run to check these two pseudo-attribute lists. The number of I/O operations performed is essentially equal to the number of records in the index file.

The remaining three audits are quite expensive in terms of I/O operations. The "$staff/$free" audit verifies those two pseudo-attribute lists by reading the entire staff file, and using identifier fields to decide whether or not each record is free or allocated. The "attribute list" audit, called only on an emergency basis, rebuilds an attribute list by reading all allocated records in the staff file. Inside the audit system, it is called by the high-level index audit. When called externally from an in-line check, the "emergency attribute list audit" performs some necessary checks before invoking the attribute list audit itself. Finally, the "staff" audit ensures that all allocated staff records are on the correct hash chain, and that the chain structure is correct. If it finds a single error on one chain, that is easily corrected; certain multiple errors on one chain force rebuilding the entire hash table, which involves reading (writing) each allocated record four times.

As may be deduced from the above, there are complicated relationships between the various parts of the audit system. Figure 3 shows a graph of the audit dependencies, that is, which audits depend upon others having previously executed. Figure 4 indicates which audits directly invoke which others on an emergency basis, as well as those invoked externally from the main system on a periodic or emergency basis. (The implicit scheduling of the quick, field, and none count audits is not shown on the figure.)

The above description might seem to imply that the audit system is able to detect and correct all errors. Of course this is not quite true. It also begs the question "who will audit the audit system?" We do not claim the audit system is perfect, nor even correct, but we wish to

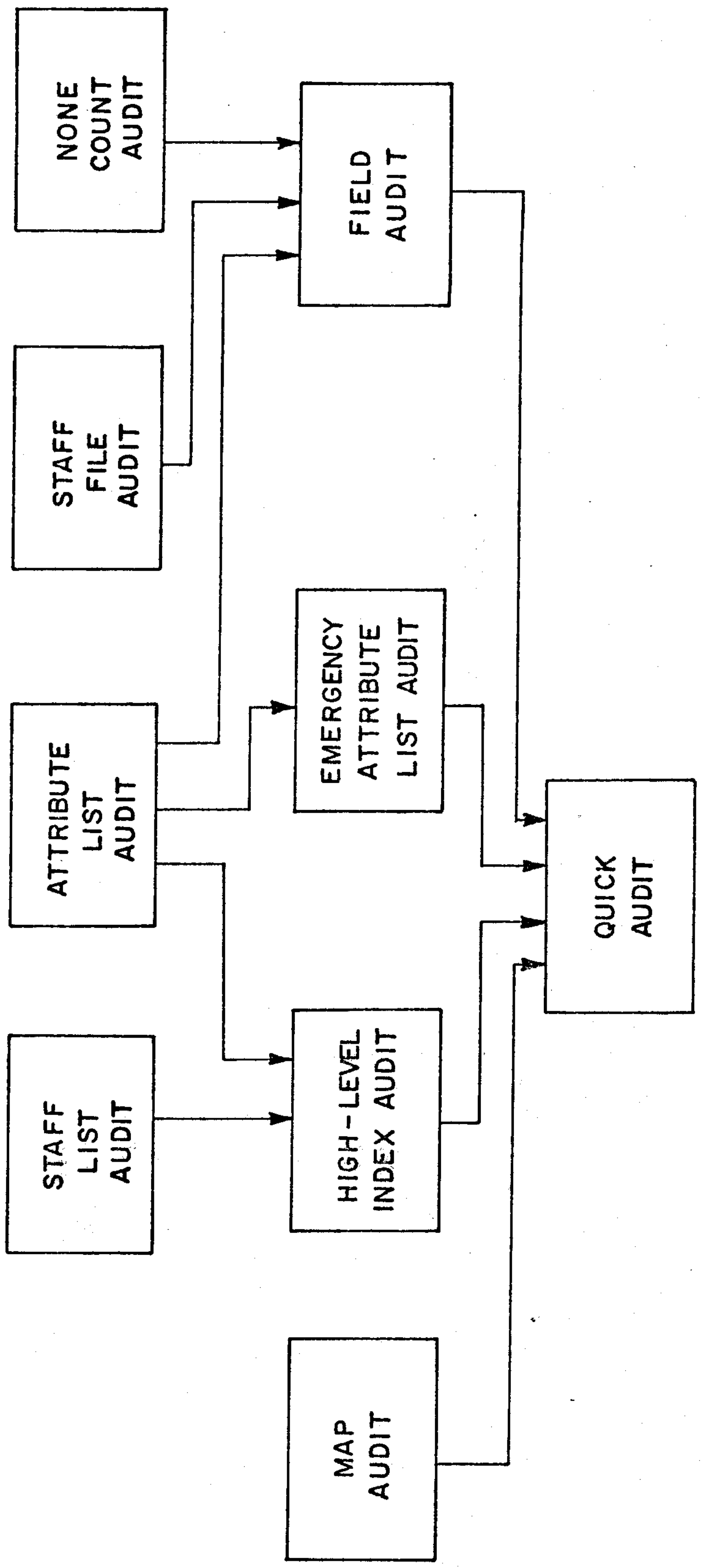

FIGURE 3: Audit Dependencies

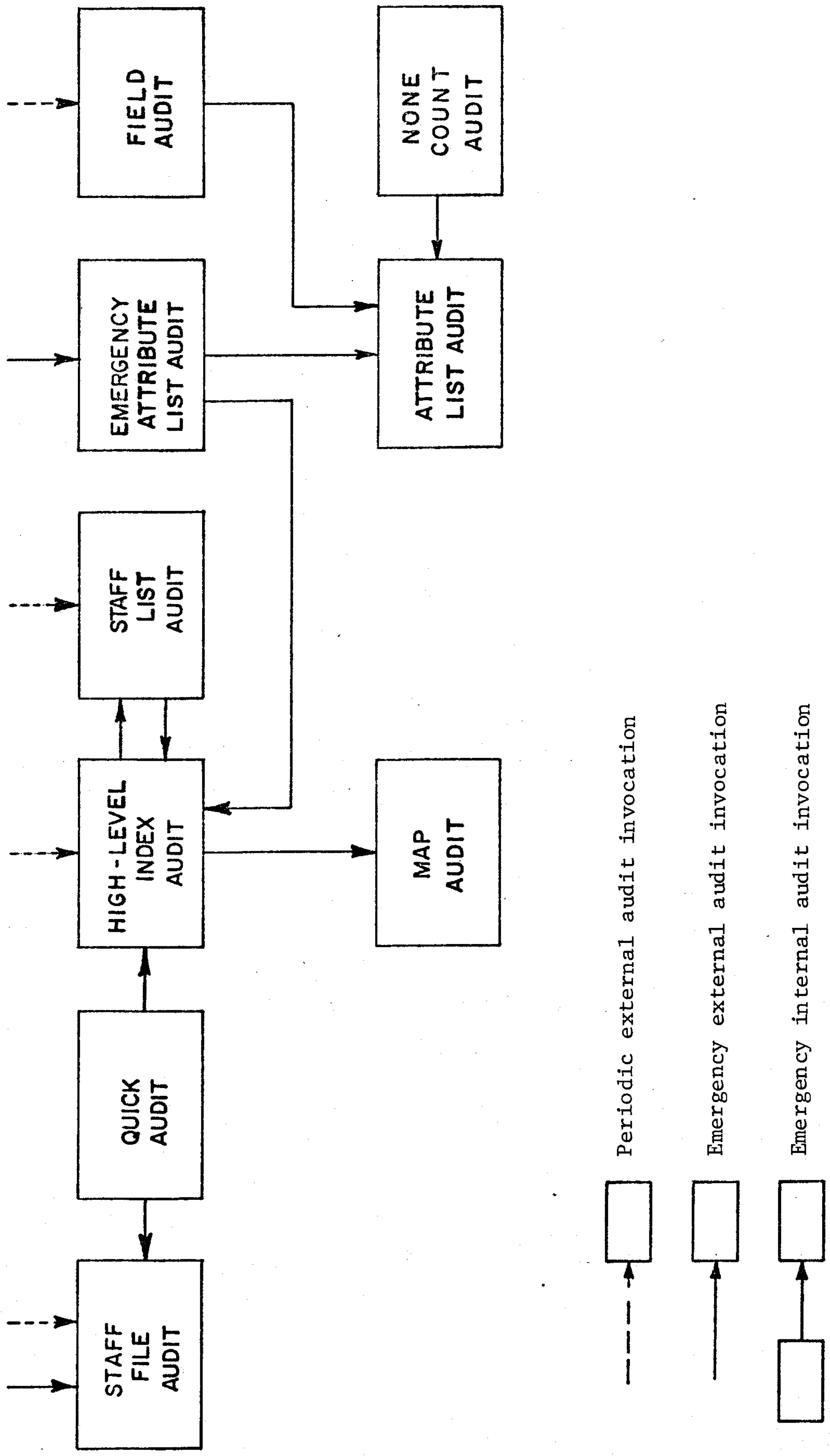
FIELD AUDIT
NONE COUNT AUDIT
EMERGENCY ATTRIBUTE LIST AUDIT
ATTRIBUTE LIST AUDIT
STAFF LIST AUDIT
HIGH-LEVEL INDEX AUDIT
MAP AUDIT
QUICK AUDIT
STAFF FILE AUDIT
Periodic external audit invocation
Emergency external audit invocation
Emergency internal audit invocation

FIGURE 4: Audit Invocation Diagram

show in the following section that it is a valuable tool which can be used to substantially increase our confidence in the reliability of the Example System. While Section 6 deals with a controlled set of experiments, a more intuitive appreciation of the audit system is given by the following incident. In one experiment, the mangler changed the key field for the master list from 1 to Ø, causing the master table to be overwritten by the master list. Although not very efficiently, and with many complaints, the audit system completely rebuilt the entire master table using the available redundancy, as well as its own knowledge of the data structure.

5. EXPERIMENTAL RESULTS

In order to be able to measure the effectiveness of our modifications, and indeed even to debug them, we required a source for errors which was both under our control, and was able to introduce errors at a rate which made experimentation feasible. A first attempt was to mangle the program by altering randomly selected conditional branches to branch on the logically opposite condition. This did not provide useful results. Eventually, we developed a "mangler", which pseudo-randomly changes records as they are written by EXSYS to external storage. We were able to specify the probability of a record's being mangled, and the probability distribution of mangles over the words of a record. As large changes to pointers and counts are often easy to detect, we designed the mangler to change selected words by small integer values, thus making the changes more subtle.

Using the mangler, a set of experiments was performed to explore the error detection and correction capabilities of Version 2 of EXSYS. Twenty experiments were performed with different random seeds used to initialise the mangler, resulting in a different set of mangles for each. A standard script was prepared, and run twice for each experiment without re-initialising the mangled files. The script contained 37 commands, and was designed so that one run of it left the files in their original state if no uncorrected mangles occurred. The last several commands tried to delete a non-existent staff record; their purpose was to increment the counters controlling audit invocation without performing write operations, thus causing some audits to be invoked. This was intended to simulate a long period of time in which all audits would be executed.

In order to facilitate the experimentation and reduce CPU time, both the frequency of mangles, and the frequency of audit invocation were set quite high. An average of 25 mangles occurred in each experiment, roughly one mangle for every three commands. The field audit (and thus the quick and none count audits) was invoked every three commands; the other periodic audits ($staff/$free, high-level index, and staff) were invoked much less often: their thresholds were set at six, but their counters were only incremented for commands which could cause a change to the related part of the data base. Full attribute list audits are only invoked on an emergency basis. Another simplification was that no mangles were permitted during audit system execution. We assumed that in any production system, the error rate would be smaller than the frequency of audit invocation, making mangles quite unlikely. The effects of a significant error rate during audit system execution require more investigation. In view of the high error rate used for the experiments, and the large number of write operations performed by the audits during correction, this was deemed impractical for the present investigation.

Each experiment produces a significant amount of information in raw form. In order to avoid obscuring EXSYS with code to interpret this information, two post-processors were written. The simpler postprocessor translates the mangler's output messages into English giving the type of record mangled, the field mangled, and the old and new field values. This was of significant help in analysing the experiments, which involved classifying the errors, and determining which had been detected, corrected, or corrected but not detected (for example, due to an attribute list being re-built). Unfortunately, we were unable to automate this part of the analysis.

The second post-processor summarises the time and I/O information produced by EXSYS. Each command or audit invocation causes a line of statistical information to be written: an identifying flag, timestamp, and accumulated I/O counts for the index file, staff file, and a work file used when intersecting attribute lists. From this, the post-processor provides a table showing CPU time and I/O operations for each type of command and audit, including minima, maxima, means, standard deviations, and overall totals.

Figure 5, which summarises the results of the experiments, requires some explanation. Serious errors are those occurring in fields essential to correct system operation as seen by the user. Redundancy errors are those in fields used only for reliability purposes. Note that the percentages of corrected and remaining errors do not sum to 100% because a small fraction of errors vanished before detection due to the normal operation of the script commands. Trivial errors (not shown) are such things as changing the name of an attribute list. Note that the mangler mangles any word in an index file record, but only structural information in a staff file record (practically all the data in index file records is structural). The "realistic" figures are obtained by deflating the frequency of audit invocation and the error rate by a factor of fifty, which we estimate as the minimum acceptable for a production

Audit Effectiveness
(Parenthesized figures are 95% confidence intervals.)

	Serious Errors (%)	Redundancy Errors (%)	All Errors (%)
Detected	86.4 (82.0, 89.8)	93.4 (88.4, 96.3)	84.2 (80.7, 87.2)
Corrected	92.4 (88.8, 94.9)	94.0 (89.2, 96.7)	87.9 (84.7, 90.5)
Remaining	5.3 (3.3, 8.5)	3.0 (1.3, 7.0)	9.7 (7.4, 12.7)

Average cost of correction:	1.03 CPU sec, 368 I/O operations
Typical command cost:	0.208 CPU sec, 58.2 I/O operations

Audit Overhead
(when no injected errors)

	Experiments		Realistic	
	CPU (sec)	I/O (opns)	CPU (sec)	I/O (opns)
Without audits	12.5	3600	628	181200
With audits	34.4	8700	712	194600
Audit overhead	175%	142%	13%	7%

Figure 5. Experimental Results

system. The time and I/O per correction figures are calculated by subtracting the time and I/O for an experimental run with no mangler from the average values with the mangler, then dividing by the average number of errors corrected.

We conclude from the experiments that we have achieved our goal of improving the Example System's fault tolerance. The experiments were also interesting for what we were able to learn from them. We gained good experience with designing a set of audits for a non-trivial system, which is no easy task in general. We discovered that writing an audit which is driven by a table which describes the structure being checked is a useful way of constructing a simpler and more easily modifiable audit. Finally, we were surprised by some of the side-effects of adding redundancy. For example, the "none counts" greatly increased our ability to detect and correct attribute list errors, but at the extreme cost of rebuilding one or more lists when the "obvious" correction was simply to adjust the count. At another extreme, incorporating the 1-correction algorithm for the double-linked hash chains applied precisely the required corrections at negligible cost compared to a complete hash chain re-build.

An interesting property of the mangler was its efficiency in pointing out coding and design problems; after some experience, we almost believed it was a malicious daemon attempting to thwart all our best efforts. Although only applicable to fault-tolerant systems, a mangler can be an extremely useful means of increasing one's confidence in a program's correctness.

6. SUMMARY, CONCLUSIONS, AND FURTHER WORK

After an introduction and some general remarks on our approach (which is only one of many) to increasing software reliability, Sections 2, 3, and 4 presented a case study involving the addition of redundancy to a set of data structures, and the use of this redundancy by a set of audit routines. Section 5 presented the results of our experimentation with EXSYS, Version 2.

We conclude that the incorporation of redundancy and audit programs can be done at an acceptable overhead cost for systems where fault tolerance is a significant requirement. While the overhead cost can be made quite small by adjusting the audit frequency, it must also be recognised that a large development expenditure is required to add redundancy and design an audit system to exploit it. This is particularly true for "complicated" data structures, where each design must presently be _ad hoc_. Additionally, there is an obvious storage overhead incurred when redundancy is added. Aside from redundant fields added to the master table and master list, and the entire set of hash table back pointers, the overhead per record was less than

10 per cent. As may be expected, one of the goals of our research is to develop increasingly general robust data structures, which can be used conveniently, and whose fault tolerance can be easily exploited through simple algorithms.

For more complicated data structures, the use of an audit scheduler eases the decompsition of the audits, especially with respect to audit priority, audit interdependency, and audit re-invocation. Another method of simplifying audit design uses tables which specify aspects of a data structure to drive a "general purpose" audit. This technique was applied to auditing the core resident tables in the quick audit, as described in Section 4. It is difficult in many cases to construct a "general purpose" audit, but it seems desirable to use this approach whenever possible.

As we have already indicated, the experimental approach described here is complemented by a growing set of theoretical results. An important class of these gives some upper and lower bounds for the detectability of an arbitrary storage structure in terms primarily of the number of disjoint sets of pointers in the structure, each of which may be used to reconstruct all structural information in an instance of it. Once the detectability is known, the General Correction Theorem [7] can be used to determine the correctability of a storage structure: a structure which has identifier fields, a stored count, is 2r-detectable, and has r + 1 edge-disjoint paths to each node of an instance is r-correctable. The theorem exhibits an algorithm which performs the correction, although its execution time is expressed in terms of a rather unfortunately large polynomial. Individually-designed correction routines usually perform more satisfactorily in practice; an example is the 1-correction routine for double-linked lists which is linear in the size of its input. Also of practical interest is a 2-detectable, 1-correctable implementation of a binary tree. All of these results are given in [5, 6, 7].

Our further work will attempt to bring the theoretical results closer and closer to being able to deal with complicated storage structures. We have defined a restricted class of compound storage structures; more general extensions in this area would be useful. Further experimentation will investigate the effects of varying the audit and mangle frequency, with the attendant increase in error propagation. Different types of mangling will also be investigated. Perhaps the most important direction for further work, however, is the inclusion of semantic information or data content in our formalism for robust data structures. General results appear difficult to obtain, but one should be able at least to include some constraint on node contents in the formalism, such as the relationship of key values to structure in a binary search tree or B-tree. Finally, it would be interesting to implement EXSYS using the recovery block techniques of Randell _et_ _al_.

In this paper, we have been concerned with the addition of redundancy to storage structures, and ways of exploiting the redundancy. Our approach was illustrated by a case study which demonstrated the costs and effectiveness of our methods. For systems requiring a degree of fault tolerance, the wise use of redundancy can bring significant benefit with a reasonable amount of overhead.

BIBLIOGRAPHY

1. R. P. Almquist, J. R. Hagerman, R. J. Hass, R. W. Peterson, and S. L. Stevens, 'Software protection in No. 1 ESS', _Proc. International Switching Symposium_, Cambridge, Mass., 565-569 (1972).

2. T. Anderson and B. Randell (eds.), _Computing Systems Reliability_, Cambridge University Press, 1979.

3. A. Avizienis. 'Fault-tolerance: The survival attribute of digital systems', _Proc. IEEE_, **66**, 1109-1125 (1978).

4. H. J. Beuscher, G. E. Gessler, D. W. Huffman, P. J. Kennedy, and E. Nussbaum, 'Administration and maintenance plan', _Bell Syst. tech J._, **48**, 2765-2815 (1969).

5. D. J. Taylor, 'Robust data structure implementations for software reliability', Ph.D. Thesis, Department of Computer Science, University of Waterloo, Ontario (1977).

6. D. J. Taylor, D. E. Morgan, and J. P. Black, 'Redundancy in data structures: Improving software fault tolerance', accepted for publication in _IEEE Trans. Software Engineering_. Also available as Computer Science Research Report CS-79-34, University of Waterloo, Ontario, Canada (1979).

7. D. J. Taylor, D. E. Morgan, and J. P. Black, 'Redundancy in data structures: Some theoretical results', accepted for publication in _IEEE Trans. Software Engineering_. Also available as Computer Science Research Report CS-79-35, University of Waterloo, Ontario, Canada (1979).

8. F. W. Tompa, 'Data structure design', _Data Structures, Computer Graphics, and Pattern Recognition_, edited by A. Klinger, et al, 3-30, Academic Press, New York (1977).

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF WATERLOO
TECHNICAL REPORTS 1979

Report No.	Author	Title
CS-79-01	E.A. Ashcroft W.W. Wadge	Generality Considered Harmful - A Critique of Descriptive Semantics
CS-79-02	T.S.E. Maibaum	Abstract Data Types and a Semantics for the ANSI/SPARC Architecture
CS-79-03	D.R. McIntyre	A Maximum Column Partition for Sparse Positive Definite Linear Systems Ordered by the Minimum Degree Ordering Algorithm
CS-79-04	K. Culik II A. Salomaa	Test Sets and Checking Words for Homomorphism Equivalence
CS-79-05*	T.S.E. Maibaum	The Semantics of Sharing in Parallel Processing
CS-79-06	C.J. Colbourn K.S. Booth	Linear Time Automorphism Algorithms for Trees, Interval Graphs, and Planar Graphs
CS-79-07	K. Culik, II N.D. Diamond	A Homomorphic Characterization of Time and Space Complexity Classes of Languages
CS-79-08	M.R. Levy T.S.E. Maibaum	Continuous Data Types
CS-79-09	K.O. Geddes	Non-Truncated Power Series Solution of Linear ODE's in ALTRAN
CS-79-10	D.J. Taylor J.P. Black D.E. Morgan	Robust Implementations of Compound Data Structures
CS-79-11	G.H. Gonnet	Open Addressing Hashing with Unequal-Probability Keys
CS-79-12	M.O. Afolabi	The Design and Implementation of a Package for Symbolic Series Solution of Ordinary Differential Equations
CS-79-13	W.M. Chan J.A. George	A Linear Time Implementation of the Reverse Cuthill-McKee Algorithm
CS-79-14	D.E. Morgan	Analysis of Closed Queueing Networks with Periodic Servers
CS-79-15	M.H. van Emden G.J. de Lucena	Predicate Logic as a Language for Parallel Programming
CS-79-16	J. Karhumäki I. Simon	A Note on Elementary Homorphisms and the Regularity of Equality Sets
CS-79-17	K. Culik II J. Karhumäki	On the Equality Sets for Homomorphisms on Free Monoids with two Generators
CS-79-18	F.E. Fich	Languages of R-Trivial and Related Monoids

* Out of print - contact author

CS-79-19	D.R. Cheriton	Multi-Process Structuring and the Thoth Operating System
CS-79-20	E.A. Ashcroft W.W. Wadge	A Logical Programming Language
CS-79-21	E.A. Ashcroft W.W. Wadge	Structured LUCID
CS-79-22	G.B. Bonkowski W.M. Gentleman M.A. Malcolm	Porting the Zed Compiler
CS-79-23	K.L. Clark M.H. van Emden	Consequence Verification of Flow-charts
CS-79-24	D. Dobkin J.I. Munro	Optimal Time Minimal Space Selection Algorithms
CS-79-25	P.R.F. Cunha C.J. Lucena T.S.E. Maibaum	On the Design and Specification of Message Oriented Programs
CS-79-26	T.S.E. Maibaum	Non-Termination, Implicit Definitions and Abstract Data Types
CS-79-27	D. Dobkin J.I. Munro	Determining the Mode
CS-79-28	T.A. Cargill	A View of Source Text for Diversely Configurable Software
CS-79-29	R.J. Ramirez F.W. Tompa J.I. Munro	Optimum Reorganization Points for Arbitrary Database Costs
CS-79-30	A. Pereda R.L. Carvalho C.J. Lucena T.S.E. Maibaum	Data Specification Methods
CS-79-31	J.I. Munro H. Suwanda	Implicit Data Structures for Fast Search and Update
CS-79-32	D. Rotem J. Urrutia	Circular Permutation Graphs
CS-79-33*	M.S. Brader	PHOTON/532/Set - A Text Formatter
CS-79-34	D.J. Taylor D.E. Morgan J.P. Black	Redundancy in Data Structures: Improving Software Fault Tolerance
CS-79-35	D.J. Taylor D.E. Morgan J.P. Black	Redundancy in Data Structures: Some Theoretical Results
CS-79-36	J.C. Beatty	On the Relationship between the LL(1) and LR(1) Grammars
CS-79-37	E.A. Ashcroft W.W. Wadge	R_x for Semantics

* Out of print - contact author

CS-79-38	E.A. Ashcroft W.W. Wadge	Some Common Misconceptions about LUCID
CS-79-39	J. Albert K. Culik II	Test Sets for Homomorphism Equivalence on Context Free Languages
CS-79-40	F.W. Tompa R.J. Ramirez	Selection of Efficient Storage Structures
CS-79-41*	P.T. Cox T. Pietrzykowski	Deduction Plans: A Basis for Intelligent Backtracking
CS-79-42	R.C. Read D. Rotem J. Urrutia	Orientations of Circle Graphs

* Out of print - contact author

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF WATERLOO
TECHNICAL REPORTS 1980

Report No.	Author	Title
CS-80-01	P.T. Cox T. Pietrzykowski	On Reverse Skolemization
CS-80-02	K. Culik II	Homomorphisms: Decidability, Equality and Test Sets
CS-80-03	J. Brzozowski	Open Problems About Regular Languages
CS-80-04	H. Suwanda	Implicit Data Structures for the Dictionary Problem
CS-80-05	M.H. van Emden	Chess-Endgame Advice: A Case Study in Computer Utilization of Knowledge
CS-80-06	Y. Kobuchi K. Culik II	Simulation Relation of Dynamical Systems
CS-80-07	G.H. Gonnet J.I. Munro H. Suwanda	Exegesis of Self-Organizing Linear Search
CS-80-08	J.P. Black D.J. Taylor D.E. Morgan	An Introduction to Robust Data Structures
CS-80-09	J.Ll. Morris	The Extrapolation of First Order Methods for Parabolic Partial Differential Equations II
CS-80-10*	N. Santoro H. Suwanda	Entropy of the Self-Organizing Linear Lists
CS-80-11	T.S.E. Maibaum C.S. dos Santos A.L. Furtado	A Uniform Logical Treatment of Queries and Updates
CS-80-12	K.R. Apt M.H. van Emden	Contributions to the Theory of Logic Programming
CS-80-13	J.A. George M.T. Heath	Solution of Sparse Linear Least Squares Problems Using Givens Rotations
CS-80-14	T.S.E. Maibaum	Data Base Instances, Abstract Data Types and Data Base Specification
CS-80-15	J.P. Black D.J. Taylor D.E. Morgan	A Robust B-Tree Implementation
CS-80-16	K.O. Geddes	Block Structure in the Chebyshev-Padé Table
CS-80-17	P. Calamai A.R. Conn	A Stable Algorithm for Solving the Multi-facility Location Problem Involving Euclidean Distances

* In preparation

CS-80-18	R.J. Ramirez	Efficient Algorithms for Selecting Efficient Data Storage Structures
CS-80-19	D. Therien	Classification of Regular Languages by Congruences
CS-80-20	J. Buccino	A Reliable Typesetting System for Waterloo
CS-80-21	N. Santoro	Efficient Abstract Implementations for Relational Data Structures
CS-80-22	R.L. de Carvalho T.S.E. Maibaum T.H.C. Pequeno A.A. Pereda P.A.S. Veloso	A Model Theoretic Approach to the Theory of Abstract Data Types and Data Structures
CS-80-23	G.H. Gonnet	A Handbook on Algorithms and Data Structures
CS-80-24	J.P. Black D.J. Taylor D.E. Morgan	A Case Study in Fault Tolerant Software
CS-80-25	N. Santoro	Four O(n**2) Multiplication Methods for Sparse and Dense Boolean Matrices
CS-80-26	J.A. Brzozowski	Development in the Theory of Regular Languages