Printing Requisition / Graphic Services

15916

1. **Please complete unshaded areas** on form as applicable.
2. **Distribute copies** as follows: White and Yellow to Graphic Services. **Retain Pink Copies for your records.**
3. On completion of order the **Yellow copy will be returned** with the printed material.
4. **Please direct enquiries, quoting requisition number and account number,** to **extension 3451.**

TITLE OR DESCRIPTION
CS-84-52 Experimentation with Data Structures

DATE REQUISITIONED: January 15, 1985
DATE REQUIRED: ASAP
ACCOUNT NO.: 4 1 2 6 8 1 1 4 1

REQUISITIONER – **PRINT**
PHONE
SIGNING AUTHORITY: J.P. Black.

MAILING INFO – NAME: Donna McCracken | DEPT.: Computer Science | BLDG. & ROOM NO.: M&C 6081E | [X] DELIVER [ ] PICK-UP

Copyright: I hereby agree to assume all responsibility and liability for any infringement of copyrights and/or patent rights which may arise from the processing of, and reproduction of, any of the materials herein requested. I further agree to indemnify and hold blameless the University of Waterloo from any liability which may arise from said processing or reproducing. I also acknowledge that materials processed as a result of this requisition are for educational use only.

NUMBER OF PAGES: 16
NUMBER OF COPIES: 100

TYPE OF PAPER STOCK
[ ] BOND [ ] NCR ____ PT. [ ] COVER [ ] BRISTOL [ ] SUPPLIED [ ] Alpaga Ivory 120M

PAPER SIZE
[ ] 8½ x 11 [ ] 8½ x 14 [ ] 11 x 17 [ ] 10X14 Glosscoat 10 pt.

PAPER COLOUR
[ ] WHITE [X] ____

INK
[X] BLACK [ ] ____

PRINTING
[ ] 1 SIDE ____ PGS. [ ] 2 SIDES ____ PGS.

NUMBERING
FROM TO

BINDING/FINISHING
[X] COLLATING [ ] STAPLING [ ] ____ HOLE PUNCHED [ ] PLASTIC RING

FOLDING/PADDING: 7X10 saddle stitch

CUTTING SIZE

Special Instructions
Beaver Cover

Both cover and inside in black ink please

COPY CENTRE — OPER. NO. | BLDG. | MACH. NO.

DESIGN & PASTE-UP — OPER. NO. | TIME | LABOUR CODE

TYPESETTING — QUANTITY | OPER. NO. | TIME | LABOUR CODE

PROOF
P R F
P R F
P R F

NEGATIVES — QUANTITY | OPER. NO. | TIME | LABOUR CODE
F L M
F L M
F L M
F L M
F L M

PMT
P M T
P M T
P M T

PLATES
P L T
P L T
P L T

STOCK

BINDERY

OUTSIDE SERVICES

$ ____ COST

TAXES – PROVINCIAL [ ] FEDERAL [ ]

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF WATERLOO

MAR 2 6 1985

1984 TECHNICAL REPORTS

No.	Authors	Title	No. of Pages	Cost of Report	Reports Requested
CS-84-32	Th. Ottmann D. Wood	Space-Economical Plane-Sweep Algorithms	20	$2.00	
CS-84-33	Th. Ottmann D. Wood	The Contour Problem for Polygons	16	2.00	
CS-84-34	F. Mavaddat	A Model for Register-Transfer Level Design Specification: The SDC Notation		2.00	
CS-84-35	C.J. Smith	A Discussion and Implementation of Kovacic's Algorithm for Ordinary Differential Equations	86	2.00	
CS-84-36	E. Chu A. George J. Liu E. Ng	SPARSPAK: Waterloo Sparse Matrix Package User's Guide for SPARSPAK-A	66	2.00	
CS-84-37	A. George E. Ng	SPARSPAK: Waterloo Sparse Matrix Package User's Guide for SPARSPAK-B	46	2.00	
CS-84-38	D. Field	Algorithms for Drawing Anti-aliased Circles and Ellipses	23	2.00	
CS-84-39	K. Culik II S. Yu	Fault Tolerant Schemes for Some Systolic Systems	29	2.00	
CS-84-40	G. Rawlins D. Wood	Computing Finitely-Oriented Polygon Intersections	7	2.00	

No.	Authors	Title	No. of Pages	Cost of Report	Report Requested
CS-84-41	J. Culberson G. Rawlins	Turtle Polygons	10	$2.00	
CS-84-42	A. George J. W. Liu	Householder Reflections versus Givens Rotations in Sparse Orthogonal Decomposition	15	2.00	
CS-84-43	A. George E. Ng	Symbolic Factorization for Sparse Gaussian Elimination with Partial Pivoting	26	2.00	
CS-84-44	J.P. Black D.J. Taylor	Local Correctability in Robust Storage Structures	23	2.00	
CS-84-45	J.P. Black D.J. Taylor	A Model for Storage Structures, Encodings, and Robustness	20	2.00	4
CS-84-46 JR	R. Lee	Concurrent Prolog and a Multi-Process Environment (ICR Report 24)	78	2.00	4
CS-84-47	M. Cheng	Design and Implementation of the Waterloo Unix Prolog Environment (ICR Report 26)	120	5.00	
CS-84-48	J.A.N. Trudel	Madame: A Planner for ISH (ICR Report 28)	213	5.00	
CS-84-49 JR	A. George M.T. Heath J.W. Liu	Parallel Cholesky Factorization on a Multiprocessor	22	2.00	4
CS-84-50	R.G. Karlsson	Algorithms in a Restricted Universe	105	5.00	
CS-84-51	J.P. Black D.J. Taylor	A Locally Correctable B-tree Implementation	18	2.00	4

No.	Author	Title	No. of Pages	Cost of Reports	Reports Requested
CS-84-52	J.P. Black D.J. Taylor	Experimentation with Data Structures	16	$2.00	1
CS-84-53	J.P. Black D.J. Taylor	Guidelines for Storage Structure Error Correction	6	2.00	
CS-84-54	F.D. Boswell D.D. Cowan T.R. Grove	A Distributed File-Server for a Personal Computer Network	20	2.00	
CS-84-55	D.D. Harms	An Investigation Into Bounds on Network Reliability	93	2.00	

TO ORDER REPORTS: Please forward the completed form, along with a cheque or money order, payable to the University of Waterloo, Department of Computer Science to:

Technical Report Secretary
Department of Computer Science
University of Waterloo
WATERLOO, Ontario, Canada
N2L 3G1

Please indicate your mailing address below:

Peter Bloniarz
Computer Science Dept.
SUNY-Albany
Albany, NY 12222
USA

rec'd $2.00 U.S.
sent report
JUN 30 [illegible]

No.	Author	Title	No. of Pages	Cost of Reports	Reports Requested
CS-84-52	J.P. Black D.J. Taylor	Experimentation with Data Structures	16	$2.00	✓
CS-84-53	J.P. Black D.J. Taylor	Guidelines for Storage Structure Error Correction	6	2.00	✓
CS-84-54	F.D. Boswell D.D. Cowan T.R. Grove	A Distributed File-Server for a Personal Computer Network	20	2.00	sent.
CS-84-55	D.D. Harms	An Investigation Into Bounds on Network Reliability	93	2.00	

TO ORDER REPORTS: Please forward the completed form, along with a cheque or money order, payable to the University of Waterloo, Department of Computer Science to:

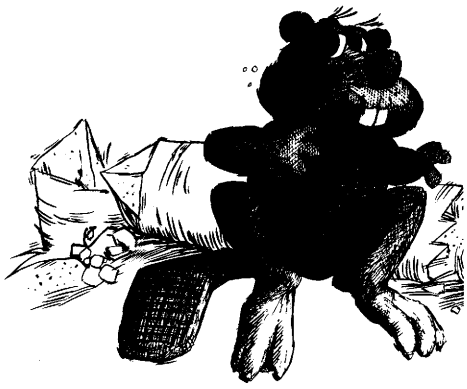
Technical Report Secretary
Department of Computer Science
University of Waterloo
WATERLOO, Ontario, Canada
N2L 3G1

Please indicate your mailing address below:

Lynn Ronas
Intel Corp.
Technical Inform. Center
2111 N.E. 25th
Hillsboro, Or. 97123

Mail 26/85
cheque for $12.00
#307 Intel Credit Union

UNIVERSITY OF WATERLOO
COMPUTER SCIENCE DEPARTMENT



Experimentation with Data Structures

D.J. Taylor
J.P. Black

Data Structuring Group
CS-84-52

December, 1984

UNIVERSITY OF WATERLOO
COMPUTER SCIENCE DEPARTMENT

Experimentation with Data Structures

David J. Taylor

James P. Black

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

ABSTRACT

Research in robust data structures can be done both by theoretical analysis of properties of abstract implementations and by empirical study of real implementations. Empirical study requires a support environment for the actual implementation. In particular, if the response of the implementation to errors is being studied, a mechanism must exist for artificially injecting appropriate kinds of errors. This paper discusses techniques used in empirical investigations of data structure robustness, with particular reference to tools developed for this purpose at the University of Waterloo.

1. INTRODUCTION

The study of robust representations of data structures has been going on at the University of Waterloo for a number of years. It is convenient to think of the activity as having three main parts: general theoretical analysis, theoretical analysis of particular structures, and empirical studies. The empirical studies are intended to investigate the response of a robust representation to "random" damage, in contrast to the theoretical analyses, which consider worst-case results.

The theoretical analyses can largely be performed using only pencil and paper, but the empirical studies require that actual update, detection, and possibly correction, procedures be implemented. In principle, this implementation could take place in any convenient programming environment. There are two significant practical problems. First, it is necessary to be able to introduce random damage into the structures being tested. This can be done in a variety of ways, but it is clearly preferable to build the "damaging" mechanisms only once and then make use of them in a variety of environments. Second, we want to be able to experiment with a wide variety of data structures, including structures

which are logically a combination of other structures. This requirement can easily lead to the duplication of much code. For example, a user interface is required for each program, and a storage structure such as "linked list" may be implemented twice—once to be used alone and once to be used as part of an externally chained hash table.

The solution to the first problem was to build a very simple input/output system which could be placed between a program manipulating data structures and the actual file system provided by the operating system. The "damaging" mechanisms were built as part of this system. The solution to the second problem was to define a standard interface for a storage structure and use this interface to allow standard support routines, such as user interface routines, to be used with many different structures. In addition, this standard interface allows general combination of structures, since storage structures which adhere to the standard interface can be "plugged into" a generic combining structure.

The software mentioned above was originally developed to help answer such questions as: given a detection procedure which can detect any one or two errors, what is the probability that it will detect a set of three errors? However, it is possible to use it for a number of other purposes. For example, in the situation just described, by running a number of one and two error cases, confidence in the correctness of the implementation can be increased. This is particularly useful with some correction procedures, which are so complex that extensive testing seems essential. In addition, by providing a standard environment, efficiency comparisons can be readily made. For example, if the average case behaviour of two alternative correction routines cannot be characterised theoretically, they may both be exercised in the same software environment, eliminating most extraneous sources of variation between the measured behaviour of the routines.

Thus, the software system described in this paper has at least three uses: determining the empirical behaviour of storage structures and associated detection or correction routines, testing certain kinds of fault tolerant software, and performance comparisons for fault tolerant software.

This paper describes the software systems introduced above, but does not report results obtained using this software, except for a few brief examples. (Some results have been published previously [2,6,8].) The paper has the following three purposes. First, to explain how results previously reported were obtained, in more detail than is possible in a paper with a different primary purpose. Second, to provide information which may be useful to others performing experiments with fault tolerant software. Third, to provide information which may be useful to others performing experiments with storage structures. While much of what we have done is related to robustness of storage structures, we believe that some methods we have developed may be applied to other forms of experimentation with storage structures.

The remainder of the paper describes the two major software systems we have developed. First, three sections describe the simple input/output system, which is known as IOSYS. The next three sections describe the standard storage structure interface and how it has been used. The resulting combination of storage structure routines and standard support routines is known as the Interchangeable Storage Structure System (ISSS). The final two sections describe

how the software has been used to perform experiments and provide a summary of the paper. The last section also contains a brief description of the historical development of the software.

2. BASIC FACILITIES

IOSYS is, in a sense, an input/output system with certain added facilities. Because the "added facilities" were the whole purpose of the implementation, an effort was made to keep the usual I/O facilities as simple as possible. Primarily as background for the following section, this section provides a brief description of the facilities provided by IOSYS when viewed as a standard input/output system.

Only one kind of file is provided by IOSYS: a fixed-length record, direct access file. Several files may be in use simultaneously with a different record length for each, but variable length records within a file are not permitted. This is a fairly restrictive environment but matches the requirements for our experimentation quite well.

Program access to files is provided by read and write functions. Files are simply identified internally by consecutive non-negative integers, so the read and write routines have a file number, a record number, and a buffer address as parameters.

During the "setup" phase of running a program under IOSYS, commands are entered from the terminal to attach the program to existing files in the "real" file system or to create new, empty files. Because many experiments are very I/O-intensive, IOSYS will normally keep small files in main storage rather than on disk.

To aid in debugging and monitoring programs, commands are provided to display records at the terminal and to activate a trace of input/output requests made during program execution.

The preceding is not a complete description of the facilities available, but does provide sufficient background for purposes of this paper.

3. MANGLING FACILITIES

In order to test the fault tolerance of software or data structures, it is necessary to have a mechanism which provides controlled simulation of the effects of faults. When studying robust data structures, the faults of interest are those which ultimately damage stored representations of data structures. Thus, a reasonable approach to the simulation of faults is to damage stored data.

In order to avoid building extensive knowledge of the data structures being used into the fault simulation mechanism, update routines can be used to guide the mechanism. Records (nodes) being accessed or updated seem logical candidates for being damaged. This consideration is the main reason that IOSYS deals with data structures which are (or at least, appear to be) on external storage: accesses and updates are forced to go through read and write function calls, thus making it easy for the fault simulation mechanism to observe such activity.

Records could be damaged (erroneously modified) when they are read or written, but since records which are being modified by an update routine seem more likely in practice to be modified erroneously, records are damaged only as they are written. At present two basic forms of damage are possible: modification of a single word in the record and refusal to write the record.

This facility for introducing damage is referred to as the "mangler." It exists as part of the IOSYS write function and is driven by a random number generator and parameters entered by the user. Since mangling is a central feature of IOSYS, a fairly detailed description of the implementation is provided here. First, there is a global control which allows the mangler to be turned on and off. Second, there is a mangle probability associated with each file. If the probability is zero, that file will not be mangled. If the mangler is on, the probability value specifies the chance of mangling on each write call.

When a mangle is to occur, the "mangle type" for the file determines what happens. Three mangle types change a single word, and two cause the record not to be written at all. The first three mangle types differ in how the word to be modified is selected: one uses a uniform distribution over all the words in the record, one uses a distribution skewed toward the beginning of the record, and one calls a user exit routine to obtain a list of "mangleable" words, from which a uniform random selection is made. Note that the last mangle type may be used to implement any arbitrary probability distribution, by selecting a single word according to this distribution and then reporting it as the only mangleable word. In each of these three cases, the record is modified by adding a value to the selected word. The value is selected uniformly from a user-specified range symmetric around zero. Adding a small quantity tends to produce more "subtle" changes than making an arbitrary replacement of the word.

The fourth mangle type simply refuses to write the record, if the probability test is satisfied. The fifth type ("crash" mangling) is intended to simulate a system crash during updating. In this case, IOSYS makes a transition from "up" to "crashed" with the specified probability. Once in "crashed" state, all writes to the file (and any other file with crash mangling specified) are refused. It is simpler to simulate crashes in this way than by attempting to abort the actual execution of an update routine. Naturally, an IOSYS call is provided to "uncrash" the system, in order to proceed to another experiment iteration, once the effects of the simulated crash have been analysed.

To allow mangling activity to be monitored, when a mangle takes place a message may be displayed at the terminal or a user exit routine may be called. The user program may request any combination of these.

4. OPTIONAL FACILITIES

The facilities described in the two previous sections are part of IOSYS itself. Modifying or replacing any of them must be done very carefully. There are other routines which work in conjunction with IOSYS, which are supplied so that code will not have to be duplicated in multiple application programs. These routines are intended to be used optionally, as required, and can be replaced by similar, user-written routines or ignored altogether, as desired.

There are presently three such packages of routines. One package provides a simple-minded free space management for IOSYS files. The other two are of greater interest.

The second package provides a "mangle table" capability. Although, in some sense, the damage done to a file must be kept secret from much of the program (for example, error detection routines clearly must not make use of such information), it is frequently important to keep a record of the mangles which have taken place. Because keeping and using such a mangle table is a non-trivial task, a package of routines is provided to do such things as: set up a mangle table, record a new mangle, find out the "true" (unmangled) value of a field, print the mangle table, and so on.

We want to make the distribution of mangles over the nodes of a structure realistic. One approach to this is to use the set of records written by an update routine as candidates for mangling rather than selecting records completely at random. This means that mangles to individual records are not independent, which seems desirable. However, using an update routine with the mangler active introduces a serious problem. If the update routine makes use of a field which has already been mangled it could propagate the damage in an unknown way, go into an infinite loop, or cause an abort. Designing update routines which will not do any of these things is an interesting problem, but in order to avoid solving the problem before performing any experiments, we wanted to use less robust update routines.

This leads to a three-file cluster for each logical file used by the program. One file is an unchanging master copy used to refresh the other two files. One of the other files is the target of actual updates, but is not mangled. The remaining file is mangled but not updated. Whenever the update routine writes a record to the update file, the corresponding record is read and rewritten in the mangle file. Since the mangler is active on this file, the write operation may result in a mangle. For efficiency, the complete update file does not really exist: only the modified records are kept in this file, a bit vector being used to indicate which records have been modified. A small set of routines is provided to handle this, so that the facility can be made conveniently available to the various different experiment programs.

5. ISSS STRUCTURE

The software described in the preceding sections was used over a period of several years to investigate the robustness of a number of structures. The structures included k-linked lists, modified(k) double-linked lists, a hash table of modified(k) double-linked lists, CTB-trees, and a very general linked list structure which included the two earlier linked list structures as special cases. (Descriptions of these storage structures have been published previously [5] and will not be repeated here.) Each of these was a separate application which used the facilities of IOSYS. While these applications were successful in accomplishing the intended experimentation, several problems were evident. (1) There was much almost-duplicate code, primarily for providing the user interface and supervising execution of experiments. When writing a new application, this code could be adapted from an existing application, but maintenance and enhancement of many near-copies is a large problem. (2) We wanted to experiment with compound structures [7]. This required a program containing one or more elementary structures and a "compound" structure which could logically combine two or more instances of such structures. Doing this on the same *ad hoc* basis as in earlier applications would take far too much effort. (3) Combining structures in other ways also seemed desirable. The hash table mentioned above is logically a combination of "hash" and "linked list." Unfortunately, linked lists had to be implemented separately for the hash table because there was then no convenient way of "plugging in" an existing linked list implementation.

In order to solve the above problems, and other similar problems, it seemed essential to be able to treat storage structures as uniform "building blocks" which could be used interchangeably. If each structure presented the same interface to a routine using it, then routines could be written to work with an arbitrary structure rather than one particular structure.

There are three entities which must be represented in a standard way: storage structures, encodings of storage structures, and instances of storage structures. The names correspond reasonably to common usage of these terms. For our purposes, a storage structure is a general implementation of some data structure, which may have parameters. For example, a linked list storage structure could implement single-linked lists, double-linked lists, etc., according to user-supplied parameters. Each storage structure is represented by a storage structure table, as described below. An encoding is a storage structure with values supplied for the parameters and which has a particular relationship to other encodings. For example, if we have a compound of two linked lists, each of the linked lists is a distinct encoding of the same storage structure. One may be single-linked and the other double-linked, but even if they have the same parameter values, one is the first component of a compound and the other is the second component of a compound. Each encoding is represented by an encoding table, as described below. An instance is a particular occurrence of some encoding. It can be specified by giving an encoding table and a sequence of header records for the instance.

It seems natural to define a storage structure implementation primarily as a collection of functions. (This notion is clearly inspired by the concept of an abstract data type [4], although our purpose is somewhat different from

conventional use of abstract data types. For example, we want to ignore, as much as possible, whether a structure requires zero keys, one keys, or many keys for insertion. In conventional use of abstract data types the number of keys would be known, only the mechanism for insertion would be hidden.) Thus, a storage structure table contains (1) the name of the structure, (2) pointers to seventeen standard functions, (3) a list of trace names, (4) a list of structure-specific commands and the functions which execute them, (5) a set of flag bits.

The name, of course, is simply used when it is necessary to identify the storage structure externally, either on input or output at the terminal.

The standard functions allow storage structures to be combined easily. They include such operations as: build an empty instance, perform insertion, perform deletion, detect and correct errors, and print instance at the terminal. Thus, for example, if a routine wants to have a certain storage structure delete a key, it can call the third function in the storage structure table for that structure, with a standard set of parameters. (Of course, the programmer doesn't even need to know that it is the third function, but uses a symbolic reference, "k_delete" in this case.) The particular seventeen functions we use were chosen because of our goals in robustness experimentation, and because of the ways we anticipated combining storage structures. A very brief description of each function is given in Table I.

The list of trace names in the storage structure table allows a central routine to handle the setting and resetting of trace bits and reporting the current status of trace bits. Otherwise, most storage structures would need to implement such a routine individually and would likely implement a less pleasant user interface. The trace bits may be used for any purpose by the storage structure routines, but generally are used to control the printing of additional information, including debugging information.

The list of structure-specific commands defines a set of commands which can be processed by this storage structure, but not necessarily by other storage structures. Putting this command list in a standard place allows a single command lookup routine to locate any general or structure-specific command. Any structure we are interested in as part of ISSS allows insertion, deletion, printing, etc., but other commands which are unique to a single structure or a small collection of structures also arise naturally. For example, each of our two B-tree implementations has a routine which, essentially, reports how full the B-tree nodes are (actually, how likely merges and balances are when keys are deleted). Making such a routine an eighteenth standard function is inappropriate since it only applies to B-trees. The scheme adopted is simply to assign an appropriate command name and place this name and a pointer to the routine in the list of structure-specific commands.

Finally, the flags in the storage structure table are used because structures are not completely interchangeable, and certain properties of a structure must be made known to other parts of the program. For example, some routines need to request deletion by specifying a record rather than a key. This is a reasonable deletion request for some structures (linked lists, binary trees) but not for others (B-trees). Thus, a flag bit is used to indicate whether the storage structure can process a deletion specified by record number.

Routine	Description
Bld_hdr	Construct an empty instance
Check	Detect and correct errors
Connected	Perform a "connectedness check"
Format	Print out the contents of a record
Freerec	Invoked when a subordinate encoding attempts to free a record
Getrec	Invoked when a subordinate encoding attempts to acquire a record
K_delete	Delete a key from an instance
K_insert	Insert a key into an instance
Print	Print an entire instance
R_delete	Delete a record, given a record number
R_insert	Insert a record, given both a key and a record number
Search	Search an instance for a key
Setup	First stage of initialisation—build encoding table
Select	Select a subordinate instance
Ud_init	Initialise data concerning which fields are candidates for introduction of errors
Ud_set	Place data in the file indicating which fields are candidates for introduction of errors
U_init	Perform initialisation immediately before beginning command processing

Table I
Standard functions for an ISSS storage structure

The second major data structure used by ISSS is the encoding table. In order to make the following description brief, some of the fields of an encoding table are omitted which are not crucial for a general understanding of ISSS. An encoding table contains (1) the index of the corresponding storage structure in the sequence of storage structures, (2) a pointer to a block of storage structure-specific parameters, (3) a word of trace flags, (4) a pointer to a vector of subordinate encoding tables and the length of that vector.

The need to find the storage structure given the encoding is obvious. The structure-specific parameters allow each structure to maintain any desired information which may vary from encoding to encoding. Examples of structure-specific parameters include the order of a B-tree, the pointer structure of a linked list, and statistical information such as the number of merges performed during B-tree deletions. The trace flags allow implementation of the trace feature mentioned above.

The arrangement of encoding tables in contiguous vectors and pointers between those vectors is used to form a tree of encoding tables. The idea is simply that elementary structures, such as lists and trees, will appear as leaves of the "encoding tree," whereas combining structures, such as compound and hash table, will appear as internal nodes. The structures combined by a combining structure appear as its immediate descendants. Thus, if we are working with a compound of two linked lists and a binary tree, the encoding tree will be as in Figure 1. (The root of the tree is a special pseudo-structure called "master," which simply exists to be the root of every encoding tree.)

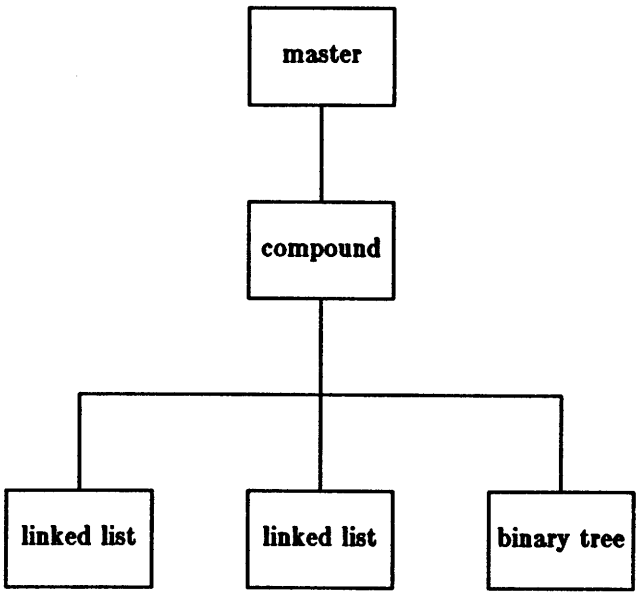


Figure 1
Encoding tree for a compound of two linked lists and a binary tree.

Thus the two central structures used by ISSS are a sequence of storage structure tables, representing all available storage structures, and an encoding tree, representing encodings currently being used.

6. ISSS STORAGE STRUCTURES

The representation of storage structures discussed in the preceding section allows storage structures to be combined easily. The purpose of this section is to give a brief overview of some ISSS storage structures which use this ability to combine structures. Of course, there are also "elementary" structures which become leaves of the encoding tree, such as linked lists, binary trees, and B-trees.

Compound storage structures were one of the original motivations for ISSS. (A compound storage structure is formed by combining two or more storage structures such that each node in the compound storage structure is a concatenation of nodes: one from each component structure [7].) A general compound structure has been implemented which allows any set of structures to be "compounded" for which compounding is logically possible. The essence of the implementation is that a compound insert operation can be performed using the insert routines of the subordinate structures, and similarly for other operations.

As another example, an externally chained hash table has been implemented. In this case, there is only one (immediately) subordinate structure, to represent the hash chains. Most of the hash table operations are essentially performed by hashing a key value, selecting the headers corresponding to the appropriate chain instance, and invoking the appropriate routine for the chain storage structure. Note that the chain need not be a linked list: it can be a binary tree, a B-tree, or a compound structure with further subordinate structures.

As a final example, we have a robust B-tree which requires a separate "level header" node for each level of the tree. Since the number of such headers will vary as the height of the tree changes, they can't be allocated statically. ISSS provides an easy method for chaining these headers together. The B-tree structure can have a linked list subordinate structure and use it to perform insertion and deletion of level headers when the tree height changes. This also makes the error detection and correction procedures for linked lists available to the B-tree implementation. Thus, the B-tree error correction procedure can begin by invoking the linked list error correction procedure to correct the level header structure, and then proceed to the correction of the B-tree itself.

7. ISSS DRIVERS

The ISSS standard interface makes it easy to implement programs which are intended to manipulate storage structures. We refer to the part of the program which is not storage structure code or ISSS general utility routines as the "driver." The effort required to implement a driver is decreased by the availability of ISSS routines for (1) command lookup, (2) interface to standard operations, (3) standard commands which are independent of storage structures.

The central command lookup actually has to handle two problems: what command is to be executed and to which instance does it apply. The second problem is resolved by requiring each combining structure to have a function which selects an instance of a subordinate structure given an appropriate token.

The user can select an arbitrary instance by starting at the root of the encoding tree and giving a sequence of tokens which successively select subordinate instances until the appropriate instance is reached. (For user convenience, there is also a "current instance" which the user can set, so a path from the root of the tree is rarely needed.) Except for the instance selection routines, individual storage structures do not need to participate in the process of deciding to which instance a command applies. Once an instance has been selected, the command can be found in one of two places: the list of commands specific to the storage structure corresponding to the selected instance, or in a general list of commands. For uniformity in performing the lookup, the general commands are actually the structure-specific commands of the "master" storage structure.

The interface routines for standard operations are fairly simple. They primarily call an appropriate routine from the storage structure corresponding to the selected instance, after setting up any special environment that routine may expect. This makes it unnecessary for each driver to handle the special requirements of the various standard functions.

Finally, there are some commands which are independent of storage structures and can be executed without invoking any storage structure routines. The trace command, described previously, is an example. Other examples are: modifying a word in a record (often useful in testing detection and correction procedures), printing a list of the commands which can be used on the current instance, printing the encoding tree. All of these are available to an arbitrary driver working with arbitrary storage structures.

ISSS has two very general drivers: the interactive driver and the experiment driver. Adding further, special-purpose, drivers is intended to be relatively easy.

The interactive driver allows a terminal user to construct, manipulate, and display storage structures. It is actually a very simple program, since almost everything which needs to be done is available as an ISSS function: the main part of the driver is a loop which (1) reads a line from the terminal, (2) calls an ISSS function to select an instance, based on the input line, (3) calls an ISSS function to look up the command verb, and (4) calls the function whose address is returned by the lookup routine.

The experiment driver allows a user to define a robustness experiment by specifying how damage is to be inserted, what checks should be performed on the damaged file, and what should be done in response to the results produced by the checks. This driver is quite complex, and would never have been implemented if it could only be used with one particular storage structure. We had been using much simpler experiment drivers which had essentially all of the experiment specification embedded in the source code. This meant that making small changes to an experiment often involved changing source code. Also, we found it necessary to have two separate experiment drivers for the CTB-tree, one for testing the detection routine and one for testing the correction routine. The next section contains some further information about the experiment driver.

As an example of a special-purpose driver, it recently became apparent that a driver was needed which would test insertion and deletion routines intensively, by invoking a detection routine after each update. It took one of the authors approximately three hours to code and test a driver for this purpose. This is

probably very close to the effort which would have been required to implement such a program for a single storage structure outside ISSS.

8. EXPERIMENTS

While it is not the purpose of this paper to report the results of specific experiments, it is appropriate to describe the kinds of experimentation and testing which can be supported.

The first type of experiment for which IOSYS and ISSS have been used is empirical detectability estimation. For example, if we know that some sets of three errors cannot be detected we would like to estimate the probability that such a set cannot be detected. To perform such an experiment we need to insert three errors, run a detection procedure, and repeat a large number of times. (For any robust data structure with detectability greater than one, it appears that the probability of introducing an undetectable set of errors is essentially zero.)

Errors can be inserted by reading and rewriting records at random, but to produce a more realistic distribution of errors, a delete routine can be used to select nodes to be written. In either case, the mangler is engaged so that some writes cause erroneous modification of the record being written. To allow all of this to be done safely and efficiently in the deletion-guided case, the three-file technique described in Section 4 is used.

A second type of experiment is a "connectedness check." In this kind of experiment, the objective is to determine empirically the probability of losing all access paths to any node (thus disconnecting the structure instance) for a given number of erroneous changes. After errors are inserted, a "connectedness checker" is run which attempts to find an access path to each node. This process is repeated some large number of times.

Another type of "experiment" is the testing of fault tolerant software. Of course, the various detection and correction routines used in the experiments described above are always tested in "trial runs" with the mangler active, in order to remove implementation bugs. It is also possible to test complicated routines, such as some correction routines, whose behaviour cannot be characterised theoretically. An example is the single error correction algorithm for CTB-trees [3]. While it is known that any single error to a CTB-tree can be corrected, this particular algorithm is so complicated that proving anything significant about it seems impossible. Therefore, it was implemented and tested, first by hand insertion of "interesting" errors. This resulted in finding a number of bugs, and appropriate modifications were made to the algorithm. When no more bugs were found by hand insertion of errors, the mangler was used to create a large number of single error test cases. These cases, even though produced completely at random, made apparent a number of bugs not previously encountered, which were then fixed. Although we cannot guarantee that any single error will be corrected on the basis of this test, we are now much more confident that the correction routine will function properly. Similar testing has subsequently been used on a number of other correction routines.

To provide a concrete example of an experiment, suppose we want to perform the first two experiments described above on a single run. Figure 2 shows the dialogue used to define the experiment. (Note that previous to the

```
mangle insertion technique?inddel
one key/record?y
verify consecutiveness?y
There are 50 keys in the instance.
Check #1?check
Check #2?connect
Check #3?
result vector?1:-
counter to be incremented?errors detected
command to be executed?
result vector?1:0
counter to be incremented?no errors detected
command to be executed?label
command to be executed?pm
command to be executed?
result vector?2:0
counter to be incremented?connected
command to be executed?
result vector?2:+
counter to be incremented?disconnected
command to be executed?
result vector?
```

Figure 2
Experiment definition dialogue with the experiment driver
(User input always follows a "?".)

```
Deletes = 16630
A total of 3000 iterations were done.
errors detected: 3000
no errors detected: 0
connected: 2071
disconnected: 929
```

Figure 3
Run summary produced by the experiment driver

material shown, the structure to be tested must be specified, as well as the number of iterations and the number of mangles. In this case, we are using a double-linked list, 3000 iterations, and three mangles per iteration.) First, we specify that "inddel" (independent deletion) should be used to insert errors. This is a deletion-based method, as discussed previously, in which all deletions start from the original instance, not the instance left by the previous delete. Next, the program asks if there is one key in each record and whether it should verify that the keys are consecutive, since "inddel" mangling requires consecutive keys. We answer "yes" to both of these questions and the system reports that there are 50 keys in the instance. The system then asks what checks are desired: we enter two, "check" which is the detection routine, and "connect" which is the connectedness checker. The "result vector?" question asks for a specification of results from some collection of checks, all of which must be matched for the associated actions to be taken. In this simple example, all the vectors are of length one. We specify four cases of interest: negative return value from check, zero return value from check, zero return value from connect, positive return value from connect. In each case, a counter will be incremented when the result vector matches, and we must assign a label to this counter. The labels used in this example indicate the meanings for the four return values above. Finally, for three of the cases, we are satisfied with only incrementing a counter, but if check returns zero, meaning it couldn't find any errors, we will be quite surprised, so for this case we request that the label of the counter and the mangle table be printed ("pm" stands for "print mangles"). Thus, each time this case occurs, "no errors detected" will be printed, followed by a mangle table. Given the labels we have assigned, the output produced at the end of the experiment is very easy to interpret, as shown in Figure 3. (The first line simply indicates how many delete operations were performed during error insertion.)

It is also possible to perform experiments intended to determine the effects of error propagation. In this kind of experiment, a script of insert and delete commands is executed with the mangler engaged. Detection and correction routines are also invoked periodically during the script execution. No measures are taken to make mangles invisible to update routines, so new errors may be introduced due to updates, and update routines may be blocked from performing any action because of encountering an error.

The objective in this kind of experiment is to determine the percentage of errors detected, percentage corrected, etc. To do this, a "mangle table" must be constructed containing data on mangles and corrections. Because of error propagation, data in this table cannot always be relied upon. Therefore, a copy of the file as it should be at the end of the script run (produced with the mangler turned off) is compared with the file actually obtained. Any differences not accounted for by the mangle table are noted. If some inserts or deletes were blocked, the number of differences noted can be very large, because a record by record comparison is not appropriate unless all updates were performed. In such cases, significant human effort is required to determine the actual number of errors detected and corrected. However, in many cases, the comparison finds no unnacounted-for differences and error detection and correction statistics produced by the program can be used immediately. At present, there is no ISSS driver for performing this type of experiment; such experiments have been performed only

for one particular storage structure, the hash table of linked lists.

9. SUMMARY AND HISTORY

In this paper, we have attempted to describe the approach we have used to the testing of robust storage structures and fault tolerant software. In order to make the discussion reasonably specific, we have included some details related to the software we have been using, but have attempted to emphasise general principles which may be useful to others intending to attempt related work. First, we described how we have used very general mechanisms for the introduction of artificial errors into data structures. Then, we described how we have been able to work with a variety of structures, including combinations of structures, without incurring excessive implementation costs.

Finally, a brief summary of their history may help to put IOSYS and ISSS in perspective. EXSYS [1] was the first instance of attempting to investigate empirical robustness of data structures. To perform EXSYS experiments, *ad hoc* mechanisms were added to the code in order to produce the necessary random damage. The result was a workable but not very convenient system. Because of the effort which would be required to perform a conversion, the current version of EXSYS still uses these *ad hoc* mechanisms.

When other empirical testing was contemplated, it seemed clear that a more general, flexible tool was required. Therefore, the first version of IOSYS was implemented, on a Honeywell 6050, in a locally designed language, Eh. The tool proved very useful, and has been modified and extended, in order to make it more powerful and useful. For various local reasons, including dropping of support for Eh, the tool was also moved to UNIX*, and translated into C.

Later, a program was implemented which allowed compound structures, composed of linked lists, to be constructed. It was intended to extend this program by adding further elementary and combining structures. Unfortunately, it became clear that the design would only allow one combining structure—compound. A major redesign of this program produced ISSS. We have gradually converted old storage structures into ISSS storage structures and implemented new storage structures under ISSS. We now have ten such structures, most of which have been mentioned as examples in this paper.

IOSYS and ISSS have now been used to perform a large number of experiments on different data structures. Although creating, modifying, and maintaining these software systems has taken a significant effort, the benefit in simplifying experimentation has easily compensated for this effort. It is our intention to go on using them for further experiments. We think IOSYS and ISSS presently have a good set of facilities for our purposes, but some extensions will likely be required to meet future needs.

It is hoped that the material presented here helps to explain how we have performed robust data structure experiments and will provide useful assistance to others who are investigating data structure robustness or testing fault tolerant software.

---

*UNIX is a trademark of Bell Telephone Laboratories.

ACKNOWLEDGEMENTS

The development of IOSYS and ISSS was possible only with the able assistance of many programmers. The authors would particularly like to acknowledge the work of Bert Bonkowski, Debbie Cullon, Ralph Hill, and Mark Ingram, who each contributed in a major way to the software development. The design of IOSYS was also influenced by helpful discussions with David Morgan. This work was supported by the Natural Sciences and Engineering Research Council of Canada under Grant A3078 and a Postgraduate Scholarship, and by the Digital Equipment Corporation.

References

1. J. P. Black, D. J. Taylor, and D. E. Morgan, "A case study in fault tolerant software," *Software--Practice and Experience* **11** pp. 145-157 (1981).
2. 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).
3. 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).
4. B. H. Liskov and S. N. Zilles, "Programming with abstract data types," *SIGPLAN Notices* **9**(4) pp. 50-59 (April 1974).
5. 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).
6. 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).
7. 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).
8. D. J. Taylor, *Robust Data Structure Implementations for Software Reliability,* Ph. D. Thesis, University of Waterloo, Ontario, Canada (August 1977).