Abstract:

The paper presents, and discusses the rationale behind, a method for structuring complex computing systems by the use of what we term "recovery blocks", "conversations" and "fault-tolerant interfaces". The aim is to facilitate the provision of dependable error detection and recovery facilities which can cope with errors caused by residual design inadequacies, particularly in the system software, rather than merely the occasional malfunctioning of hardware components.

System structure for software fault tolerance

By

B. Randell

TECHNICAL REPORT SERIES

Series Editor: Dr. B. Shaw

Number 75
May, 1975

© 1975 University of Newcastle upon Tyne.

Printed and published by the University of Newcastle upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU.
RANDELL, Brian

System structure for software fault tolerance.
[By] B. Randell.

Newcastle upon Tyne: University of Newcastle upon Tyne Computing Laboratory, 1975.

(University of Newcastle upon Tyne Computing Laboratory, Technical Report Series, No. 75).

12"

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE, Computing Laboratory.
Technical Report Series. 75.

Suggested classmarks (primary classmark underlined)

Library of Congress:
Dewey (17th): 001.64404 621.381952
U.D.C.: 681.322.06/.02

Suggested keywords

ACCEPTANCE TEST
ALTERNATE BLOCK
CHECKPOINT
CONVERSATION
ERROR DETECTION
ERROR RECOVERY
RECOVERY BLOCK
RECURSIVE CACHE

Abstract

The paper presents, and discusses the rationale behind, a method for structuring complex computing systems by the use of what we term "recovery blocks", "conversations" and "fault-tolerant interfaces". The aim is to facilitate the provision of dependable error detection and recovery facilities which can cope with errors caused by residual design inadequacies, particularly in the system software, rather than merely the occasional malfunctioning of hardware components.

About the Author

Professor Randell is Professor of Computing Science in the Computing Laboratory of Newcastle upon Tyne University.
SYSTEM STRUCTURE FOR SOFTWARE FAULT TOLERANCE

R. Randell
Computing Laboratory,
University of Newcastle upon Tyne, U.K.

Keywords
Error detection, Error Recovery, Recovery Block, Recursive Cache, Checkpoint, Acceptance Test, Alternate Block, Conversation.

Abstract
The paper presents, and discusses the rationale behind, a method for structuring complex computing systems by the use of what we term "recovery blocks", "conversations" and "fault-tolerant interfaces". The aim is to facilitate the provision of dependable error detection and recovery facilities which can cope with errors caused by residual design inadequacies, particularly in the system software, rather than merely the occasional malfunctioning of hardware components.

Contents
1. Introduction
2. Fault Tolerance in Software
3. Recovery Blocks
   3.1 Acceptance Tests
   3.2 Alternates
   3.3 Restoring the System State
4. Error Recovery amongst Interacting Processes
   4.1 Process Conversations
5. Multi-Level Systems
   5.1 Errors above a Virtual Machine Interface
   5.2 Errors below a Virtual Machine Interface
   5.3 Fault-Tolerant Virtual Machine Interfaces
6. Conclusions
7. Acknowledgements
8. References.

1. Introduction

The concept of 'fault-tolerant computing' has existed for a long time. The first book on the subject [10] was published no less than ten years ago, but the notion of fault tolerance has remained almost exclusively the preserve of the hardware designers. Hardware structures have been developed which can 'tolerate' faults, i.e. continue to provide the required facilities despite occasional failures, either transient or permanent. Internal component failures are only one source of unreliability in computing systems, decreasing in significance as component reliability improves, while software faults have become increasingly prevalent with the steadily increasing size and complexity of software systems.

In general, fault-tolerant hardware designs are expected to be correct — i.e., the tolerance applies to component failures rather than design inadequacies — although the dividing line between the two may on occasion be difficult to define. But all software faults result from design errors. The relative frequency of such errors reflects the much greater logical complexity of the typical software design compared to that of a typical hardware design. The difference in complexity arises from the fact that the 'machines' that hardware designers produce have a relatively small number of distinctive internal states, whereas the designer of an even a small software system has, by comparison, an enormous number of different states to consider — thus one can usually afford to try hardware designs as being "correct", but often cannot do the same with software even after extensive validation efforts. (The difference in scale is evidenced by the fact that a software simulator of a computer, written at the level of detail required by the hardware designers to analyze and validate their logical design, is usually one or more orders of magnitude smaller than the operating system supplied with that computer.)

If all design inadequacies could be avoided or removed this would suffice to achieve software reliability. We here use the term 'design' to include 'implementation', which is actually merely low-level design, concerning itself with detailed design decisions whose correctness nevertheless can be as vital to the correct functioning of the software as that of any high-level design decision.) Indeed many writers equate the terms "software reliability" and "program correctness", however, until reliable correctness proofs (relative to some correct and adequately detailed specification), which cover even implementation details, can be given for systems of a realistic size, the only alternative means of
increasing software reliability is to incorporate provisions for software fault tolerance.

In fact there exist sophisticated computing systems, designed for environments requiring near-
continuous service, which contain ad hoc checks and checkpointing facilities that provide a measure
of tolerance against some software errors as well as hardware failures [1]. They incidentally
demonstrate the fact that fault tolerance does not necessarily require diagnosing the cause of the
fault, or even deciding whether it arises from the hardware or the software. However there has been
comparatively little specific research into techniques for achieving software fault tolerance,
and the constraints they impose on computing system design.

It was considerations such as these that led to the establishment at the University of
Newcastle upon Tyne of a project on the design of highly reliable computing systems, under the
sponsorship of the Science Research Council of the United Kingdom. The aims of the project were and
are “to develop, and give a realistic demonstration of the utility of, computer architecture and
programming techniques which will enable a system to have a very high probability of continuing to
give a trustworthy service in the presence of hardware faults and/or software errors, and during
their repair. A major aim will be to develop techniques which are of general utility, rather than
limited to specialised environments, and to explore possible trade-offs between reliability
and performance”.

A modest number of reports and papers have emanated from the project to date, including a
general overview [2], papers concerned with addressing and protection [6,7], and a preliminary
account of our work on error detection and recovery [5]. The present paper endeavours to provide a
rather more extensive discussion of our work on system error recovery techniques, and concentrates
on techniques for system structuring which facilitate software fault tolerance. A companion paper
[1] presents a proof-guided methodology for designing the error detection routines that our method
requires.

2. Fault Tolerance in Software

All fault tolerance must be based on the provision of useful redundancy, both for error detection
and error recovery. In software the redundancy required is not simple replication of programs but
redundancy of design.

The scheme for facilitating software fault tolerance that we have developed can be regarded
as analogous to what hardware designers term ‘stand-by sparing’. As the system operates, checks are
made on the acceptability of the results generated by each component. Should one of these checks
fail, a spare component is switched in to take the place of the erroneous component. The spare
component is of course not merely a copy of the main component. Rather it is of independent
design, so that there can be hope that it can cope with the circumstances that caused the main
component to fail. (These circumstances will comprise the data the component is provided with
and, in the case of errors due to faulty process synchronisation, the timing and form of its
interactions with other processes.)

In contrast to the normal hardware stand-by sparing scheme, the spare software component is
invoked to cope with merely the particular set of circumstances that resulted in the failure of the
main component. We assume the failure of this component to be due to residual design inadequacies,
and hence that such failures occur only in exceptional circumstances. The number of different
sets of circumstances that can arise even with a software component of comparatively modest size is
immense. Therefore the system can revert to the use of the main component for subsequent operations
– in hardware this would not normally be done until the main component had been repaired.

The variety of undetected errors which could have been made in the design of a non-trivial
software component is essentially infinite. Due to the complexity of the component, the relationship
between any such error and its effect at run time may be very obscure. For these reasons we
believe that diagnosis of the original cause of software errors should be left to humans to do,
and should be done in comparative leisure. Therefore our scheme for software fault tolerance in
no way depends on automated diagnosis of the cause of the error – this would surely result only
in greatly increasing the complexity and therefore the error-proneness of the system.

The recovery block scheme for achieving software fault tolerance by means of stand-by sparing
has two important characteristics:

(i) it incorporates a general solution to the problem of switching to the use of the spare
component, i.e., of repairing any damage done by the erroneous main component, and of
transferring control to the appropriate spare component.

(ii) it provides a method of explicitly structuring the software system which has the effect
of ensuring that the extra software involved in error detection and in the spare components
does not add to the complexity of the system, and so reduce rather than increase overall
system reliability.

3. Recovery Blocks

Although the basic recovery block scheme has already been described elsewhere [5], it is
convenient to include a brief account of it here. We will then describe several extensions to the
scheme directed at more complicated situations than the basic scheme was intended for. Thus we
start by considering the problems of fault tolerance, i.e., of error detection and recovery,
within a single sequential process in which assignments to stored variables provide the only means
of making recognisable progress. Considerations of the problems of communication with other
processes, either within the computing system (e.g. by a system of passing messages, or the use
of shared storage) or beyond the computing system (e.g. by explicit input/output statements) is
deferred until a later section.
The progress of a program is by its execution of sequences of the basic operations of the computer. Clearly, error checking for each basic operation is out of the question. Apart from questions of expense, absence of an awareness of the wider scene would make it difficult to formulate the checks. We must aim at achieving a tolerable quantity of checking and exploit our knowledge of the functional structure of the system to distribute these checks to best advantage. It is standard practice to structure the text of a program of any significant complexity into a set of blocks (by which term we include subroutine, procedure, subroutines, paragraph, etc.) in order to simplify the task of understanding and documenting the program. Such a structure allows one to provide a functional description of the purpose of the program text constituting a block. (This text may of course include calls on subsidiary blocks.) The functional description can then be used elsewhere in place of the detailed design of the block. Indeed, the structuring of the program into blocks, and the specification of the purpose of each block, is likely to precede the detailed design of each block, particularly if the programming is being performed by more than one person.

When executed on a computer, a program which is structured into blocks evokes a process which can be regarded as being structured into operations. Operations are seen to consist of sequences of smaller operations, the smallest operations being those provided by the computer itself. Our scheme of system structuring is based on the selection of a set of these operations to act as units of error detection and recovery, by providing extra information with their corresponding blocks, and so turning the blocks into recovery blocks.

The scheme is not dependent on the particular form of block structuring that is used, or the rules governing the scopes of variables, methods of parameter passing, etc. All that is required is that when the program is executed the acts of entering and leaving each operation are explicit, and that operations are properly nested in time. (In addition, although it is not required, considerable advantage can be taken of information which is provided indicating whether any given variable is local to a particular operation.) However, for convenience of presentation, we will assume that the program text is itself represented by a nested structure of Algol or PL/I-style blocks.

A recovery block consists of a conventional block which is provided with a means of error detection (an acceptance test) and zero or more stand-by spares (the additional alternates). A possible syntax for recovery blocks is as follows:

```<recovery block> ::= <enure <acceptance test> by <program text> <other alternate> else error

<primary alternate> ::= <alternate> <other alternate> else error

<other alternate> ::= <empty> <other alternate> else by <alternate>

<alternate> ::= <statement list>
```

The primary alternate corresponds exactly to the block of the equivalent conventional program, and is entered to perform the desired operation. The acceptance test, which is a logical expression without side-effects, is evaluated on exit from any alternate to determine whether the alternate has performed acceptably. A further alternate, if one exists, is entered if the preceding alternate fails to complete (e.g. because it attempts to divide by zero, or exceeds a time limit), or fails the acceptance test. However before an alternate is so entered, the state of the process is restored to that current just before entry to the primary alternate. If the acceptance test is passed, any further alternates are ignored, and the statement following the recovery block is the next to be executed. However if the last alternate fails to pass the acceptance test then the entire recovery block is regarded as having failed, so that the block in which it is embedded fails to complete, and recovery is then attempted at that level.

```<acceptance test> ::= <logical expression>
```

Consider the recovery block structure shown in Figure 2. The acceptance test BT will be invoked on completion of primary alternate BP. If the test succeeds, the recovery block B is left and the program text immediately following is reached. Otherwise the state of the system is reset and alternate EQ is entered. If EQ, and then BT, do not succeed in passing the acceptance test the recovery block B as a whole, and therefore primary alternate BP, are regarded as having failed. Therefore the state of the system is reset even further, to that current just before entry to AP, and alternate AQ is attempted.

Deferring for the moment questions as to how the state of the system is reset when necessary, the recovery block structure can be seen as providing a very general framework for the use of stand-by sparing which is in full accordance with the characteristics discussed earlier, in section 2. There is no need for, indeed no possibility of, attempts at automated error diagnosis because of the fact that the system state is reset after
since test incorporates a check that the set of items in S after operation of an alternate are indeed in order. However, rather than incur the cost of checking that these elements are a permutation of the original items, it merely requires the sum of the elements to remain the same.

\[ \text{ensure } \text{sorted}(S) \land (\text{sum}(S) = \text{sum}(\text{prior} S)) \]

by quicksort \(S\) else by bubblesort \(S\) else error

**Fig. 3:** A fault-tolerant sort program

### 3.2 Alternates

The primary alternate is the one which is intended to be used normally to perform the desired operation. Other alternates might attempt to perform the desired operation in some different manner, presumably less economically, and preferably more simply. Thus, as long as one of these alternates succeeds the desired operation will have been completed, and only the error log will reveal any troubles that occurred.

However, in many cases one might have an alternate which performs a less desirable operation, but one which is still acceptable to the enclosing block in that it will allow the block to continue properly. (One plentiful source of both these kinds of alternate might be earlier releases of the primary alternate.)

Figure 4 shows a recovery block consisting of a variety of alternates. (This figure is taken from Anderson [1].) The aim of the recovery block is to extend the sequence 5 of items by a further item \(i\), but the enclosing program will be able to continue even if afterwards \(S\) is merely "consistent". The first two alternates actually try, by different methods, to join the item \(i\) onto the sequence \(S\). The other alternates make increasingly desperate attempts to produce at least some sort of consistent sequence, providing appropriate warnings as they do so.

\[ \text{ensure } \text{consistent sequence}(S) \]

by extend \(S\) with \(i\) else by concatenate to \(S\) (construct sequence \(i\)) else by warning ("lost item") else by \(S\) = construct sequence \(i\); warning ("correction, lost sequences") else by \(S\) = empty sequence; warning ("lost sequence and items") else error

**Fig. 4:** A recovery block with alternates which achieve different but still acceptable though less desirable results.

### 3.3 Restoring the System State

By making the resetting of the system state completely automatic, the programmers responsible for designing acceptance tests and alternates are shielded from the problems of this aspect of error recovery. No special restrictions are placed on the operations which are performed within the alternates, on the calling of procedures or the modification of global variables, and no special programming conventions have to be adhered to. In particular the error-prone task of explicit preservation of restart information is avoided. It is thus that the recovery block structure provides a framework which enables extra program text to be added to a conventional program, for purposes of specifying error detection and recovery actions, with good reason to believe that despite the increase in the total size of the program its overall reliability will be increased.

All this depends on being able to find a method of automating the resetting of the system state whose overheads are tolerable. Clearly, taking a copy of the entire system state on entry to each recovery block, though in theory satisfactory, would in normal practice be far too inefficient. Any method involving the saving of sufficient information during program execution for the program to be executable in reverse, instruction by instruction, would be similarly impractical.

Whenever a process has to be backed up, it is to the state it had reached just before entry to the primary alternate - therefore the only values that have to be reset are those of non-local variables that have been modified. Since no explicit restart information is given, it is not known beforehand which non-local variables should be saved. Therefore we have designed various versions of a mechanism which arranges that non-local variables are saved in what we term a "recursive cache" as and when it is found that this is necessary, i.e. just before they are modified. The mechanisms do this by detecting, at run time, assignments to non-local variables, and in particular by recognising when an assignment to a non-local variable is the first to have been made to that variable within the current alternate. Thus precisely sufficient information can be preserved.

The recursive cache is divided into regions, one for each nested recovery level, i.e. for each recovery block that has been entered and not yet left. The entries in the current cache region will contain the prior values of any variables that have been modified within the current recovery block, and thus in case of failure it can be used to back up the process to its most recent recovery point. The region will be discarded in its entirety after it has been used for backing up a process; however, if the recovery block is completed successfully some cache entries will be discarded, but those that relate to variables which are non-local to the enclosing environment will be consolidated with those in the underlying region of the cache.

A full description of one version of the mechanism has already been published [5], so we will not repeat this description here. We envisage that the mechanism would be at least partly built in hardware, at any rate if, as we have assumed here, recovery blocks are to be provided within ordinary programs working on small data items such as scalar variables. If however one were programming solely in terms of operations on large blocks of data, such as entire arrays or files, the overheads caused by a mechanism built completely from software would probably be supportable.

Indeed the recursive cache scheme, which is essentially a means for correctly preventing what is sometimes termed "update in place", can be viewed...
as a generalisation of the facility in CAP's "middleware" scheme [6] for preventing individual application programs from destructively updating files.

The various recursive cache mechanisms can all work in terms of the basic unit of assignment of the computer - e.g., a 32 bit word. Thus they ensure that just those scalar variables and array elements which are actually modified are saved. It would of course be possible to structure a program so that all its variables are declared in the outermost block, and within each recovery block each variable is modified, and so require that a maximum amount of information be saved. In practice we believe that even a moderately well-structured program will require comparatively little space for saved variables. Measurements of space requirements will be made on the prototype system now being implemented, but already we have some evidence for this from some simple experiments carried out by interpretively executing a number of Algol W programs. Even regarding each Algol block as a recovery block it was found that the amount of extra space that would be needed for saved scalar variables and array elements was in every case considerably smaller at all times than that needed for the ordinary data of the program.

The performance overheads of the different recursive cache mechanisms are in the process of being evaluated. Within a recovery block only the speed of store instructions is affected, and once a particular non-local variable has been saved subsequent stores to that variable take place essentially at full-speed. The overheads involved in entering and leaving recovery blocks differ somewhat between the various mechanisms, but two mechanisms incur overheads which depend just linearly on the number of different non-local variables which are modified. It is our assessment that these overheads will also be quite modest. Certainly it would appear that the space and time overheads incurred by our mechanisms will be far smaller than would be incurred by any explicitly programmed scheme for saving and restoring the process state.

4. Error Recovery among Interacting Processes

In the mechanism described so far, the only notion of forward progress is that of assignment to a variable. In order to reset the state of a process after the failure of an acceptance test, it was necessary only to undo assignments to non-local variables. In practice, however, there are many other ways of making forward progress during computations - e.g., positioning a disk arm or magnetic tape, reading a card, printing a line, receiving a message, or obtaining real-time data from external sensors - but already actions are difficult or even impossible to undo. However, their effects must be undone in order not to compromise the inherent "recoverability" of state provided by the recursive cache mechanisms.

Our attempt to cope with this kind of problem is based on the observation that all such forms of progress involve interaction among processes. In some cases, one or more of these processes may be mechanical, human, or otherwise external - e.g., the process representing the notion of the card-

reading machinery. In other cases, the progress can be encapsulated in separate but interacting computational processes, each of which is structured by recovery blocks. In this section, we will explore the effect of this latter type of interaction on the backtracking scheme, still restricting each process to simple assignment as the only method of progress. Then in section 5, we will explore the more general problem.

Consider first the case of two or more interacting processes which have the requirement that if one attempts to recover from an error, then the others must also take recovery action, to keep in step.

For example, if one process fails after having received, and destroyed, information from another process, it will require the other process to resupply this information. Similarly a process may have received and acted upon information subsequently discovered to have been sent to it in error, and so must abandon its present activity. Maintaining, naturally, our insistence on the dangers of attempted programmed error diagnosis, we must continue to rely on automatic backtracking of processes to the special recovery points provided by recovery block entries. Each process while executing will at any moment have a sequence of recovery points available to it, the number of recovery points being given by the level of dynamic nesting of recovery blocks.

An isolated process could "use up" recovery points just one at a time by suffering a whole series of over more serious errors. However given an arbitrary set of interacting processes, each with its own private recovery structure, a single error on the part of just one process could cause all the processes to use up many or even all of their recovery points, through a sort of uncontrolled domino effect.

The problem is illustrated in Figure 5, which shows three processes, each of which has entered four recovery blocks that it has not yet left. The dotted lines indicate the interactions between processes (i.e., an information flow resulting in an assignment in at least one process). Should process 1 now fail, it will be backed up to its latest i.e., its fourth recovery point, but the other processes will not be affected. If process 2 fails, it will be backed up to its fourth recovery point past an interaction with process 1, which must therefore also be backed up to the recovery point immediately prior to this interaction, i.e., its third recovery point. However if process 3 fails, all the processes will have to be backed up right to their starting points.

![Fig. 5: The domino effect](image-url)
The domino effect can occur when two particular circumstances exist in combination:

(i) the recovery block structures of the various processes are unco-ordinated, and take no account of process interdependencies caused by their interactions.

(ii) the processes are symmetrical with respect to failure propagation – either member of any pair of interacting processes can cause the other to back up.

By removing either of these circumstances, one can avoid the danger of the domino effect. Our technique of structuring process interactions into 'conversations', which we describe next, is a means of dealing with point (i) above: the concept of multi-level processes, described in section 5 of this paper, will be seen to be based on avoiding symmetry of failure propagation.

4.1 Process Conversations

If we are to provide guaranteed recoverability of a set of processes which by interacting have become mutually dependent on each other's progress, we must arrange that the processes co-operate in the provision of recovery points, as well as in the interchange of ordinary information. To extend the basic recovery block scheme to a set of interacting processes, we have to provide a means of co-ordinating the recovery block structures of the various processes, in effect to provide a recovery structure which is common to the set of processes. This structure we term a conversation.

Conversations, like recovery blocks, can be thought of as providing firewalls (in both time and space) which serve to limit the damage caused to a system by errors. Figure 6 represents this view of a recovery block as providing a firewall for a single process. The downward pointing arrow represents the overall progress of the process. The top edge of the recovery block represents the environment of the process on entry, which is preserved automatically and can be restored for the use of an alternate block. The bottom edge represents the acceptable state of the process on exit from the recovery block, as checked by the acceptance test, and beyond which it is assumed that errors internal to the recovery block should not propagate. Of course the strength of this firewall is only as good as the rigour of the acceptance test.) The sides show that the process is isolated from other activities, i.e., that the process is not subject to external influences which

![Fig. 6: A recovery block in a single sequential process.](image)

cannot be recreated automatically for an alternate, and that it does not generate any results which cannot be suppressed should the acceptance test be failed. (These side firewalls are provided by some perhaps quite conventional protection mechanism, to complement the top and bottom firewalls provided by the recursive cache mechanism and acceptance test.)

The manner in which the processing is performed within the recovery block is of no concern outside it, provided that the acceptance test is satisfied. For instance, as shown in figure 7, the process may divide into several parallel processes within the recovery block. The recursive cache mechanisms that we have developed permit this, and place no constraints on the manner in which this parallelism is expressed, or on the means of communication between those parallel processes.

Any of the parallel processes could of course enter a further recovery block, as shown in Fig. 8. However, by doing so it must lose the ability to communicate with other processes for the duration of its recovery block. To see this, consider the consequences of an interaction between the processes at points E and F. Should process I now fail its acceptance test it would resume at point A with an alternate block. But there is no

![Fig. 7: Parallel processes within a recovery block.](image)

![Fig. 8: Parallel processes within a recovery block, with a further recovery block for one of the processes. Interaction between the processes at points E and F must now be prohibited.](image)
Fig. 14, to show the main features of a fault-tolerant interface of the complete interpreter kind. For purposes of comparison, Fig. 14 shows the equivalent interface in a conventional complete interpreter.

The basic difference between a fault-tolerant interpreter and a conventional interpreter is that, for each different type of instruction to be interpreted, the fault-tolerant interpreter, in general, provides a set of three related procedures rather than just a single procedure. The three procedures are as follows:

(i) an interpretation procedure — this is basically the same as the single procedure provided in a conventional interpreter, and provides the normal interpretation of a particular type of instruction. But within the procedure, the interface ensures that before any changes are made to the state of the interpreted process or the values of any of its variables, a test is made to determine whether any information should first be saved in order that fall back will be possible.

(ii) an inverse procedure — this will be called when a process is being backed up, and will make use of information saved during any uses of the interpretation procedure.

(iii) an acceptance procedure — this will be called when an alternate block has passed its acceptance test, and allows for any necessary tidying up and checking related to the previous use of the normal interpretation procedure.

When the instruction is one that does not change the system state, inverse and acceptance procedures are not needed. If the instruction is, for example merely a simple assignment to a scalar, the interpretation procedure saves the value and the address of the scalar before making the first assignment to the scalar within a new recovery block. The inverse procedure uses this information to reset the scalar, and there is a trivial acceptance procedure. A non-trivial acceptance procedure would be needed, if, for example, the interpreter had to close a file and perhaps do some checking on the file information in order to complete the work stemming from the use of the interpretation procedure.

A generalization of the recursive cache, as described in section 3.3, is used to control the invocation of inverse and acceptance procedures. The cache records the descriptors for the inverse and acceptance procedures corresponding to interpretation procedures that have been executed and caused system state information to be saved. Indeed each cache region can be thought of as containing a linear "program", rather than just a set of saved prior values. The "program" held in the current cache region indicates the sequence of inverse procedures calls that are to be "executed" in order to back up the process to its most recent recovery point. (If the process passes its acceptance test the procedure calls in the "program" act as calls on acceptance procedures.) The program of inverse/acceptance calls is initially null but grows as the process performs actions which add to the task of backing it up. As with the basic recursive cache mechanism, the cache region will be discarded in its entirety after it has been used for backing up a process. Similarly, if the recovery block or conversation is completed successfully, some entries will be discarded, but those that relate to variables which are non-local to the enclosing environment will be consolidated with the existing "program" in the underlying region of the cache.

This then is a very brief account, ignoring various simple but important "mere optimizations", of the main characteristics of a failure-tolerant virtual machine interface of the complete interpreter kind. Being so closely related to the basic recursive cache mechanism, it will perhaps be most readily appreciated by people who are already familiar with the published description [5] of the detailed functioning of one recursive cache mechanism.

6. Conclusions

The techniques for structuring fault-tolerant systems which we have described have been designed especially for faults arising from design errors, such as are at present all too common in complex software systems. However we believe they are also of potential applicability to hardware and in particular allow the various operational faults that hardware can suffer from to be treated as simple special cases. In fact the techniques we have sketched for fault tolerance in multi-level systems would appear to provide an appropriate means of integrating provisions for hardware reconfiguration into the overall structure of the system. Indeed as a general approach to the structuring of a complex activity where the possibility of errors is to be considered, there seems to be no a priori reason why the structuring should not extend past the confines of the computer system. Thus, as others have previously remarked [2], the structuring could apply to the environment and perhaps even the activity of the people surrounding the computer system.
The effectiveness of this approach to fault-tolerant system design will depend critically on the acceptance tests and additional alternate blocks that are provided. An experimental prototype system is currently being developed which should enable us to obtain experience in the use of this approach, to evaluate its merits, and to explore possible performance/reliability trade-offs. In our opinion one lesson is however already clear. If it is considered important that a complex system be provided with extensive error recovery facilities whose dependability can be the subject of plausible a priori arguments, then the system structure will have to conform to comparatively restrictive rules. Putting this another way, it will not be sufficient for designers to argue for the use of very sophisticated control structures and intercommunication facilities on the grounds of performance characteristics and personal freedom of design, unless they can clearly demonstrate that these do not unduly compromise the recoverability of the system.

7. Acknowledgements

The work reported in this paper is the result of the efforts of a sizeable number of people, all associated with the Science Research Council-sponsored project on system reliability at the University of Newcastle upon Tyne. Arising out of this work, patents have been applied for by the National Research Development Corporation. Those most directly responsible for the techniques of system structuring which form the central theme of this paper are Jim Herning, Ron Herr, Hugh Lauer, Mike Melliar-Smith and the author. (Professor Herning, of the University of Toronto, held a Senior Visiting Fellowship with the project during the Summer of 1971.)

The author has great pleasure in acknowledging his indebtedness to all his colleagues on the project, several of whom, and in particular Mike Melliar-Smith, provided many useful acceptance tests and alternates for the author's activity in preparing this paper.

8. References


