System Structure for Software Fault Tolerance

BRIAN RANDELL

Abstract — This 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.

Index Terms — Acceptance test, alternate block, checkpoint, conversation, error detection, error recovery, recovery block, recursive cache.

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 designer. Hardware structures have been developed which can "tolerate" faults, i.e., continue to provide the required facilities despite occasional failures, either transient or permanent, of internal components and modules. However, hardware 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 even a small software system has, by comparison, an enormous number of different states to consider — thus one can usually afford to treat 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 [11]. 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 tradeoffs between reliability and performance." A modest number of reports and papers have emanated from the project to date, including a general overview [12], papers concerned with addressing and protection [6], [7], and a preliminary account of our work on error detection and recovery [5]. The present paper endeavors 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.

II. 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 synchronization, 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 nontrivial 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.

1) 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.

2) 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.

III. 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 recognizable 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 module, procedure, subroutine, 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> :: =	**ensure** <acceptance test> **by**
	<primary alternate>
	<other alternates> **else error**
<primary alternate> :: =	<alternate>
<other alternates>.:: =	<empty> | <other alternates >
	else by <alternate>
<alternate> :: =	<statement list>
<acceptance test> :: =	<logical expression>

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.

In the illustration of a recovery block structure in Fig. 1, double vertical lines define the extents of recovery blocks, while single vertical lines define the extents of alternate blocks, primary or otherwise. Fig. 2 shows that the alternate blocks can contain, nested within themselves, further recovery blocks.

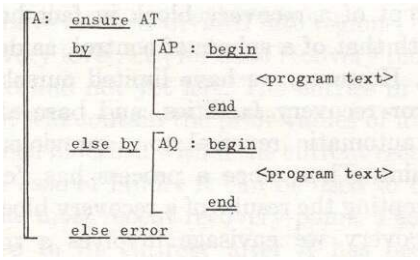

Fig. 1. Simple recovery block.

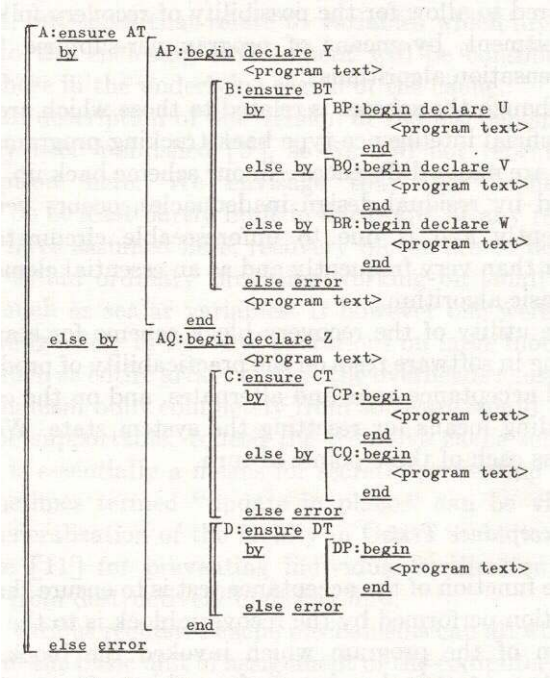

Fig. 2. More complex recovery block.

Consider the recovery block structure shown in Fig. 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 BQ is entered. If BQ and then BR do not succeed in passing the acceptance test the recovery block B as a whole, and therefore primary alternate AP, 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 II. 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 an error, deleting all effects of the faulty alternate. Once the system state is reset, switching to the use of an alternate is merely a matter of a simple transfer of control.

The concept of a recovery block in fact has much in common with that of a sphere of control, as described by Davies [2]. However, we have limited ourselves to preplanned error recovery facilities, and base all error recovery on automatic reversal to a previously reached recovery point. Thus, once a process has "committed" itself by accepting the results of a recovery block, the only form of recovery we envisage involves a more global process reversal, to the beginning of a recovery block whose results have not yet been accepted. In contrast, Davies is prepared to allow for the possibility of recovery following commitment, by means of programmer-supplied "error compensation algorithms."

Although the scheme is related to those which are used in artificial intelligence-type back-tracking programs [4], there are major differences — in our scheme back up, being caused by residual design inadequacies, occurs very infrequently and is due to unforeseeable circumstances, rather than very frequently and as an essential element of the basic algorithm.

The utility of the recovery block scheme for stand-by sparing in software rests on the practicability of producing useful acceptance tests and alternates, and on the cost of providing means for resetting the system state. We will discuss each of these points in turn.

A. Acceptance Tests

The function of the acceptance test is to ensure that the operation performed by the recovery block is to the satisfaction of the program which invoked the block. The acceptance test is therefore performed by reference to the variables accessible to that program, rather than variables local to the recovery block, since these can have no effect or significance after exit from the block. Indeed the different alternates will probably have different sets of local variables. There is no question of there being separate acceptance tests for the different alternates. The surrounding program may be capable of continuing with any of a number of possible results of the operation, and the acceptance test must establish that the results are within this range of acceptability, without regard for which alternate can generate them.

There is no requirement that the test be, in any formal sense, a check on the absolute "correctness" of the operation performed by the recovery block. Rather it is for the designer to decide upon the appropriate level of rigor of the test. Ideally the test will ensure that the recovery block has met all aspects of its specification that are depended on by the program text that calls it — in practice, if only for reasons of cost and/or complexity, something less than this might have to suffice. (A methodological approach to the design of appropriate acceptance tests is described by Anderson [1].)

Although when an acceptance test is failed all the evidence is hidden from the alternate which is then called, a detailed log is kept of such incidents, for off-line analysis. Some failures to pass the acceptance test may be spurious because a design inadequacy in the acceptance test itself has caused an unnecessary rejection of the operation of an alternate. In fact the execution of the program of the acceptance test itself might suffer an error and fail to complete. Such occurrences, which hopefully will be rare since the aim is to have acceptance tests which are much simpler than the alternates they check, are treated as failures in the enclosing block. Like all other failures they are also recorded in the error log. Thus the log provides a means of finding these two forms of inadequacy in the design of the acceptance test — the remaining form of inadequacy, that which causes the acceptance of an incorrect set of results, is of course more difficult to locate.

```
ensure sorted (S) ∧ (sum(S) = sum(prior S)
by quickersort (S)
else by quicksort (S)
else by buddlesort (S)
else error
```

Fig. 3. Fault-tolerant sort program.

When an acceptance test is being evaluated, any nonlocal variables that have been modified must be available in their original as well as their modified form because of the possible need to reset the system state. For convenience and increased rigor, the acceptance test is enabled to access such variables either for their modified value or for their original (prior) value. One further facility available inside an acceptance test will be a means of checking whether any of the variables that have been modified have not yet been

accessed within the acceptance test — this is intended to assist in detecting sins of commission, as well as omission, on the part of the alternate.

Fig. 3 shows a recovery block whose intent is to sort the elements of the vector S. The acceptance 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.

B. 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 alternates might be earlier releases of the primary alternate!)

Fig. 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 S 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.

```
ensure 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 sequence")
else by S:= empty sequence; warning ("lost
          sequence and item")
else error
```

Fig. 4. Recovery block with alternates which achieve different, but still acceptable though less desirable, results

C. 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 nonlocal variables that have been modified. Since no explicit restart information is given, it is not known beforehand which nonlocal variables should be saved. Therefore we have designed various versions of a mechanism which arranges that nonlocal 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 nonlocal variables, and in particular by recognizing when an assignment to a nonlocal 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 nonlocal 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 secretly preventing what is sometimes termed "update in place," can be viewed as a generalization of the facility in CAP's "middleware" scheme [11] 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 nonlocal 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 nonlocal 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.

IV. ERROR RECOVERY AMONGST 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 nonlocal 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. These 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 motion 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 V 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 backing up 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 ever 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.

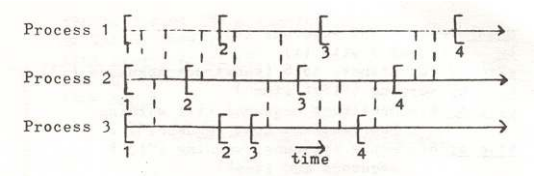

Fig. 5. Domino effect

The problem is illustrated in Fig. 5, which shows three processes, each of which has entered four recovery blocks that it has not yet left. The dotted lines indicate 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!

The domino effect can occur when two particular circumstances exist in combination.

1) The recovery block structures of the various processes are uncoordinated, and take no account of process interdependencies caused by their interactions.

2) 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 1) above; the concept of multilevel processes, described in Section V of this paper, will be seen to be based on avoiding symmetry of failure propagation.

A. 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 cooperate 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 for coordinating 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.

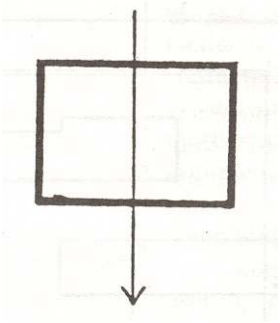

Fig. 6. Recovery block in a single sequential process.

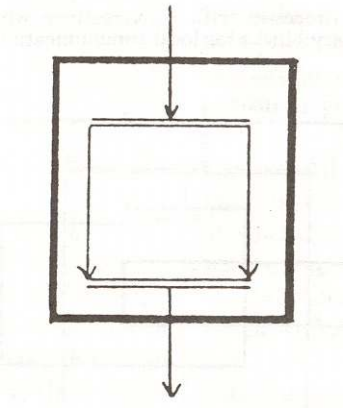

Fig. 7. Parallel processes within a recovery block.

Fig. 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 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 Fig. 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 these parallel processes.

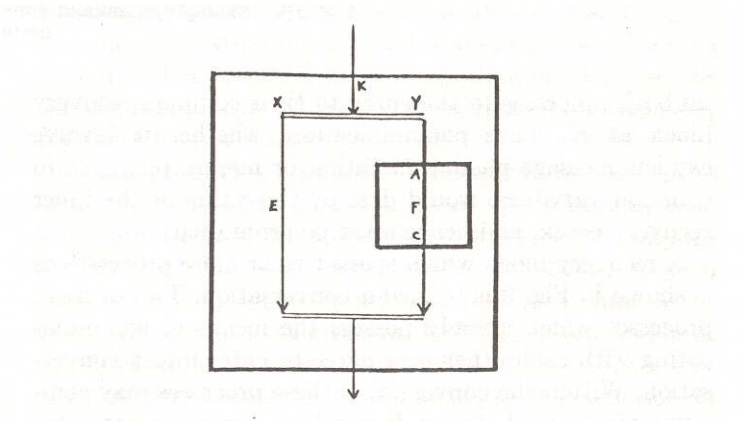

Fig. 8. Parallel processes within a recovery block, with a further recovery block for one of the processes. Interaction between the processes at points E and F must now be prohibited.

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 Y now fail its acceptance test it would resume at point A with an alternate block. But there is no way of causing process X to repeat the interaction at E without backing up both processes to the entry to their common recovery block at K. Thus communication, whether it involve explicit message passing facilities, or merely reference to common variables, would destroy the value of the inner recovery block, and hence must be prohibited.

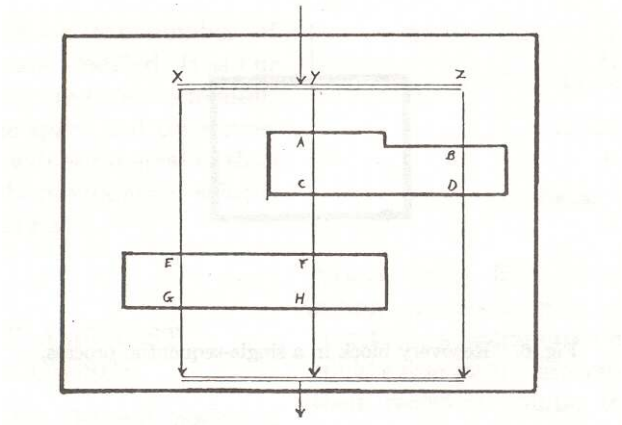

Fig. 9. Parallel processes with conversations which provide recovery blocks for local communication.

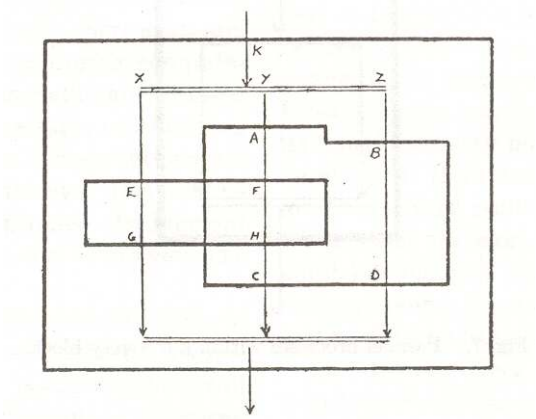

Fig. 10. Example of invalid conversations which are not strictly nested.

A recovery block which spans two or more processes as is shown in Fig. 9 is termed a conversation. Two or more processes which already possess the means of communicating with each other may agree to enter into a conversation. Within the conversation these processes may communicate freely between themselves, but may not communicate with any other processes. At the end of the conversation all the processes must satisfy their respective acceptance tests and none may proceed until all have done so. Should any process fail, all the processes must automatically be backed up to the start of the conversation to attempt their alternates.

As is shown in Fig. 9, it is possible that the processes enter a conversation at differing times. However all processes must leave the conversation together, since no process dare discard its recovery point until all processes have satisfied their respective acceptance tests. In entering a conversation a process does not gain the ability to communicate with any process with which it was previously unable to communicate — rather, entry to a conversation serves only to restrict communication, in the interests of error recovery. As with recovery blocks, conversations can of course occur within other conversations, so as to provide additional possibilities for error detection and recovery. However conversations which intersect and are not strictly nested cannot be allowed. Thus structures such as that shown in Fig. 10 must be prohibited, as can be demonstrated by an argument similar to that given in relation to Fig. 8.

V. MULTILEVEL SYSTEMS

We turn now to a method of structuring systems which uses assymetrical failure propagation in order to avoid the uncontrolled domino effect described in Section IV. In so doing we extend the scope of our discussions to cover more complex means of making recognizable progress than simple assignments. Moreover, we also face for the first time the possibility of reliability problems arising from facilities used to provide the means of constructing and executing processes and of using recovery blocks and conversations. The method of structuring which permits these extensions of our facilities for fault tolerance involves the use of what we (and others) term multilevel systems.

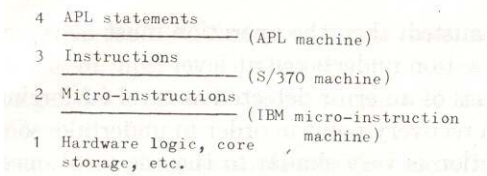

Fig. 11. Fully interpretive multilevel system.

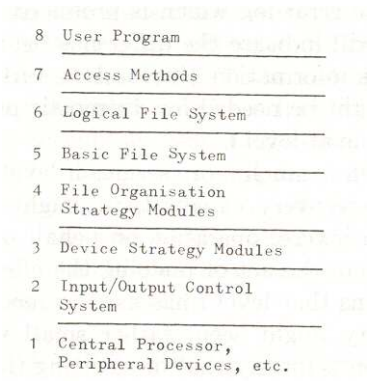

Fig. 12. Multilevel file system interpretive only at level 1 (see Madnick and Alsop [8]).

A multilevel system is characterized by the existence of a sequence of defined "abstract" or "virtual" machines which denote the internal interfaces between the various levels. A given virtual machine provides a set of apparently atomic facilities (operations, objects, resources, etc.). These can be used to construct the set of facilities that constitute a further (higher) virtual machine interface, possibly of a very different appearance. Each virtual machine is therefore an abstraction of the virtual machine below it. Since we are concerning ourselves with computer systems, we in general expect each virtual machine to have the characteristics of a programmable computer. Thus it is capable of executing a program that specifies which operations are to be applied to which operands, and their sequencing.

Our use of the term virtual machine is quite general. In particular our concept of multilevel systems includes systems whose levels are entirely different from each other (as in Fig, 11) as well as systems whose levels have much in common with each other (as in Fig. 12), for example being constructed by applying a protection scheme on a single computer. However, in each case the operations that a given virtual machine provides can be regarded as atomic at the level above it and as implemented by the activity of the level immediately below the virtual machine interface. Thus from the viewpoint of level ***i*** of the system in Fig. 12, the whole of the file accessing operation is performed by level 7. Indeed even the operation of addition, and the whoJe process of instruction fetching and decoding, can be regarded as being provided by level 7. This is the case no matter which actual level below level 7 is in fact responsible for the construction of these facilities out of more basic ones.

Some virtual machine interfaces allow the facilities they provide to be used without much, or even any, knowledge of the underlying structures used to construct these facilities. Virtual machine interfaces which have this characteristic can be termed opaque interfaces. Such virtual machine interfaces are total (in the sense that a mathematical function which is defined for all possible arguments is total) and have associated documentation which completely defines the interface. Being total and completely documented are necessary rather than sufficient conditions for a virtual machine interface to be usefully opaque, a characteristic which only well-chosen ones possess in any great measure, but this is a subject which we will not pursue further here.

Opaque virtual machine interfaces facilitate the understanding of existing complex systems, and the design of new ones. They do this by enabling the complexity of the system to be divided and conquered, so that no single person or group of persons has to master all the details of the design. They can therefore in themselves contribute to the overall reliability of a system, by simplifying the tasks of its designers. However, if design errors are made, or operational failures of physical components occur, it will be found that existing methods of constructing opaque virtual machine interfaces are somewhat inadequate. The sought-after opacity of the interface will in many cases be lost, since error recovery (either manual or predesigned) will need an understanding of two or more levels of the system. Hence our interest in providing facilities for tolerating faults, including those due to design errors, which can be used by designers whose detailed understanding of the system is limited to that of a single level and the two virtual machine interfaces that bound it. (A very different approach to these problems, based on the use of programmer-supplied error diagnosis and recovery code, has been described by Parnas [9].)

All this presupposes that the virtual machine interfaces have some physical realization in the operational system. Conceptual levels, though of value during system design and in providing documentation of the behavior of a reliable system, typically play no part in failure situations — for example the levels in the THE system [3] have no relevance to the problem of coping with, say, an actual memory parity error. The actual physical realization in existing multilevel systems can vary widely — from, for example, the provision of physically separate storage and highways for microprograms and programs, to the use of a single control bit to distinguish between supervisor and user modes of instruction execution. What we now describe are additional general characteristics and facilities that we believe any such physical realization of a virtual machine interface should possess in order to support our techniques for system fault tolerance.

A. Errors Above a Virtual Machine Interface

Everything that appears to happen in a given level is in fact the result of activity for which the level below is (directly or indirectly) responsible. This applies not only to the ordinary operations performed at a level but also to any recovery actions which might be required. Consider for example a level ***i*** which uses our recovery block scheme to provide itself with some measure of fault tolerance, and which makes recognizable progress by means of simple assignment statements. Then it is level ***i-1*** which is responsible not only for the actual assignments, but also for any saving of prior values of variables and reinstatement of them when required.

Similarly, if the virtual machine which supports level ***i*** includes any more exotic operations which change the system state as seen by level ***i***, e.g., magnetic tape rewind, then level ***i-1*** will have the responsibility of undoing their effects, e.g., repositioning the tape (whether level ***i-1*** undertakes this responsibility itself, or instead delegates it to level ***i-2*** is irrelevant).

Provided that level ***i-1*** fulfills its responsibilities level ***i*** can thus assume that error detection will automatically be followed by a return to the most recent recovery point. This will occur whether the detection of a level ***i*** error occurs at level ***i*** itself (e.g., by means of an acceptance test) or below level ***i*** because of incorrect use by level ***i*** of one of the operations provided to it by level ***i-1*** (e.g., division by zero).

It should be noted that both progress and fall back, as recognizable in the level above a virtual machine interface, are provided by progress on the level below, i.e., the level ***i-1*** keeps going forwards, or at least tries to, even if it is doing so in order to enable level ***i*** to (appear to) go backwards.

For example, level ***i*** might read cards from an "abstract card reader" while level ***i-1*** actually implements this abstract card reader by means of spooling. When level ***i*** encounters an error and tries to go backwards, it must appear to "unread" the cards read during the current recovery block. But level ***i-1*** implements this "unreading" by merely resetting a pointer in its spool buffer — a positive or forward action on its part.

All this assumes level ***i-1*** is trouble free — what we must now discuss are the complications caused by level ***i-1*** being unable, for various reasons, to maintain its own progress, and in particular that progress on which level ***i*** is relying.

B. Errors Below a Virtual Machine Interface

Needless to say, the programs which provide a virtual machine interface can themselves, if appropriate, incorporate recovery blocks for the purpose of local error detection and recovery. Thus when level ***i-1*** makes a mistake, which is detected, while performing some operation for level ***i***, if an alternate block manages to succeed where the primary alternate had failed the operation can nevertheless be completed. In such circumstances the program at level ***i*** need never know that any error occurred. (For example, a user process may be unaware that the operating system had to make several attempts before it succeeded in reading a magnetic tape on behalf of the user process.) But if all the alternates of the outermost recovery block of the level ***i-1*** program performing an operation for level ***i*** fail, so that the recovery capability at level ***i-1*** is exhausted, then the operation must be rejected and recovery action undertaken at level ***i***.

This case of an error detected at level ***i-1*** forcing level ***i*** back to a recovery point in order to undertake some alternative action is very similar to the one mentioned earlier in Section V-A — namely that of an error detected at level ***i-1***, but stemming from the incorrect use of an operation by level ***i***. The error log which is produced for later offline analysis will indicate the difference between the two cases, but this information

(leave alone further information which might be needed for diagnostic purposes) will not be available at level ***i***.

The situation is much more serious if level ***i-1*** errs, and exhausts any recovery capability it might have, whilst performing an inverse operation on behalf of level ***i***, i.e., fails to complete the act of undoing the effects of one or more operations that level ***i*** has used to modify its state. This possibility might seem rather small when the inverse operation is merely that of resetting the prior value of a scalar variable. However when an inverse operation is quite complex (e.g., one that involves undoing the changes a process has caused to be made to complicated data structures in a large filing system) one might have to cope with residual design inadequacies, as well as the ever-present possibility of hardware failure,

When an inverse operation cannot be completed, the level ***i*** cannot be backed up, so it has to be abandoned. This is perhaps the most subtle cause for level ***i-1*** to abandon further attempts to execute a level ***i*** process — more familiar ones include the sudden inability of level ***i-1*** to continue fetching and decoding level ***i*** instructions, locating level ***i*** operands, etc., either because of level ***i-l***'s own inadequacy, or that of the level ***i-2*** machine on which it depends. (For example, level 3 of Fig. 11, the APL interpreter, might find that the file in which it keeps the APL program belonging to a particular user was unreadable, a fault which perhaps was first detected at level 2, by the microprogram.) .

There is one other important class of errors detected below a virtual machine interface which can be dealt with without necessarily abandoning level ***i***, the level above the interface. After level ***i*** has passed an acceptance test, but before all the information constituting its recovery point has been discarded, there is the chance for level ***i-1*** to perform any checking that is needed on the overall acceptability, in level ***i-1*** terms, of the sequence of operations that have been carried out for level ***i***.

For example, level ***i*** may have been performing operations which were, as far as it was concerned, disk storage operations. Level ***i-1*** could in fact have buffered the information so stored. Before the present level of fall back capability of level ***i*** is discarded, level ***i-1*** may wish to ensure that the information has been written to disk and checked. If level ***i-1*** finds that it cannot ensure this, but instead encounters some problem from which it itself is unable to recover, then it can in essence cause level ***i*** to fail, and to fall back and attempt an alternate. This will be in the hope that whatever problem it was that level ***i-1*** got into (on behalf of level ***i***) this time, next time the sequence of operations that level ***i*** requests will manage to get dealt with to the satisfaction of level ***i-1*** as well as of level ***i***.

In fact an interesting example of this case of level ***i-1*** inducing a failure in level ***i*** occurs in the mechanization of conversations. Consider a level ***i*** process which is involved in a conversation with some other level ***i*** process and which after completing its primary alternate satisfies its acceptance test. At this moment level ***i-1*** must determine whether the other process has also completed its primary alternate and passed its acceptance test. If necessary the process must be suspended until the other process has been completed, as discussed in Section IV-A. If the other process should fail, then the first process must also be forced to back up just as if it had failed its own acceptance test even though it had in fact passed it.

C. Fault-Tolerant Virtual Machine Interfaces

We have so far discussed the problems of failures above and below a virtual machine interface quite separately. In fact, except for the highest level and the one that we choose to regard as the lowest level, every level is of course simultaneously below one virtual machine interface and above another such interface. Therefore each interface has the responsibility for organizing the interaction between two potentially unreliable levels in a multilevel system. The aim is to embody within the interface all the rules about interaction across levels that we have been describing, and so simplify the tasks of designing the levels on either side of the interface.

If this can be done then it will be possible to design levels which are separated by opaque virtual machine interfaces independently of each other, even in the case where the possibility of failures is admitted. By enabling the design of error recovery facilities to be considered separately for different levels of the system, in the knowedge that the fault-tolerant interface will arrange their proper interaction, their design should be greatly simplified — a very important consideration if error recovery facilities in complex systems are to be really relied upon.

Various different kinds of virtual machine interfaces are provided in current multilevel systems. These range from an interface which involves complete interpretation (e.g., the APL machine interface in Fig. 11 and the lowest interface in Fig. 12), to one where many of the basic facilities provided above the interface are in fact the same as those made available to the level immediately below the interface by some yet lower virtual machine interface (e.g., the other interfaces in Fig. 12). These latter kinds of interface, because of their performance characteristics, can be expected to predominate in systems which have many levels — in theory the multilevel file system (Fig. 12) could be built using a hierarchy of complete interpreters, but this is of course wildly impractical.

It is not appropriate within the confines of this already lengthy paper to give a fully detailed description of even a single kind, leave alone the various different kinds, of fault-tolerant virtual machine interface. However we have attempted, with Fig. 13, 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.

1) *An Interpretation Procedure:* This is basically the same as the single procedure provided in a conventional interpreter, and provides the normal interpretation of the 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.

2) *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.

3) *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 nontrivial acceptance procedure would be needed if, for example, the interpreter had to close a file and perhaps do some checking on the filed 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 III-C, 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 nonlocal to the enclosing environment will be consolidated with the existing "program" in the underlying region of the cache.

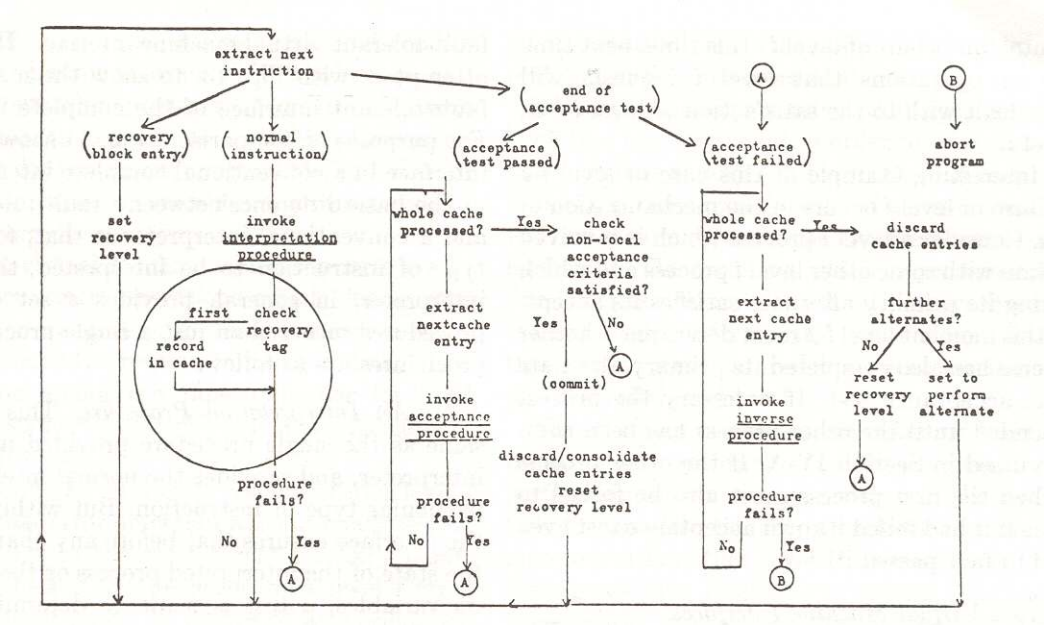

Fig. 13. Fault-tolerant interpreter.

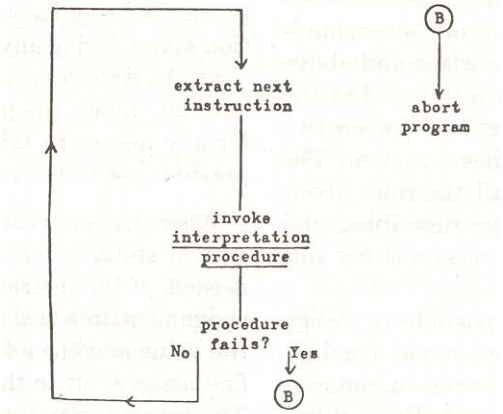

Fig. 14. Conventional interpreter.

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.

VI. 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 multilevel 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 systems 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 tradeoffs. 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.

ACKNOWLEDGMENT

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 J. J. Horning, R. Kerr, H. C. Lauer, P. M. Melliar-Smith, and the author. (Professor Horning, of the University of Toronto, Toronto, Ont., Canada, held a Senior Visiting Fellowship with the project during the Summer of 1973.) The author has great pleasure in acknowledging his indebtedness to all his colleagues on the project, several of whom, and in particular P. M. Melliar-Smith, provided many useful acceptance tests and alternates for the author's activity in preparing this paper.

REFERENCES

[1] T. A. Anderson, "Provably safe programs," Comput. Lab., Univ. Newcastle upon Tyne, Newcastle upon Tyne, England, Tech. Rep., in preparation.

[2] C. T. Davies, "Recovery semantics for a DB/DC system," in *Proc. 1973 Ass. Comput. Mach. Nat. Conf.*, New York, N. Y., pp. 136-141.

[3] E. W. Dijkstra, "The structure of the "THE" multiprogramming system," *Commun. Ass. Comput. Mach.*, vol. 11, no. 5, Rp. 341-346, 1968.

[4] C. Hewitt, "PLANNER: A language for proving theorems in robots," in *Proc. Int. Joint Conf. Artificial Intelligence*, Mitre Corp., Bedford, Mass., 1969, pp. 295-301.

[5] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, and B. Randell, "A program structure for error detection and recovery," in *Proc. Conf. Operating Systems: Theoretical and Practical Aspects*, IRIA, Apr. 23-25, 1974, pp. 177-193.

[6] H. C. Lauer, "Protection and hierarchical addressing structures," in *Proc. Int. Workshop Protection in Operating Systems*, IRIA, Rocquencourt, France, 1974, pp. 137-148.

[7] H. C. Lauer and D. Wyeth, "A recursive virtual machine architecture," Comput. Lab., Univ. Newcastle upon Tyne, Newcastle upon Tyne, England, Tech. Rep. 54, Sept. 1973.

[8] S. E. Madnick and J. W. Alsop, II, "A modular approach to file system design," in *1969 Spring Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 34. Montvale, N. J.: AFIPS Press, 1969, pp. 1-13.

[9] D. L. Parnas, "Response to detected errors in well-structured programs," Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, Pa., Tech. Rep., July 1972.

[10] W. H. Pierce, *Failure-Tolerant Computer Design*. New York: Academic, 1965.

[11] B. Randell, "Highly reliable computing systems," Comput. Lab., Univ. Newcastle upon Tyne, Newcastle upon Tyne, England, Tech. Rep. 20, July 1971.

[12] ----, "Research on computing system reliability at the University of Newcastle upon Tyne, 1972/73," Comput. Lab., Univ. Newcastle upon Tyne, Newcastle upon Tyne, England, Tech. Rep. 57, Jan. 1974.

Brian Randell was born in Cardiff, Wales, on April 26, 1936. He received the B.Sc. degree in mathematics from the University of London, London, England, in 1957. From 1957 to 1964 he was employed with the English Electric Company Limited, where he worked on compilers for DEUCE and the KDF9, and was responsible for the Whetstone KDF9 compiler. From 1964 to 1969 he was employed with IBM working mainly at the IBM T. J. Watson Research Center, Yorktown Heights, N. Y. During that time he worked on the architecture of very high speed computers, multi-processing systems, and system design methodology. Since 1969 he has been Professor of Computing Science at the University of Newcastle upon Tyne, Newcastle upon Tyne, England. He is currently the principal investigator for the project on the design of highly reliable computing systems sponsored by the Science Research Council; co-author of the book *ALGOL 60 Implementation* (New York: Academic, 1964); and Editor of the book *The Origins of Digital Computers: Selected Papers* (New York: Springer-Verlag, 1973).

Mr. Randell is a member of the Association for Computing Machinery and a Fellow of the British Computer Society.