FAULT TOLERANT COMPUTING SYSTEMS

B. Randell

Computing Laboratory, University of Newcastle upon Tyne

ABSTRACT

Fault tolerance involves the provision of strategies for error detection, damage assessment, fault treatment and error recovery. A survey is given of the different sorts of strategies used in highly reliable computing systems, together with an outline of recent research on the problems of providing fault tolerance in parallel and distributed computing systems.

1. INTRODUCTION

System reliability is sometimes interpreted rather broadly as a measure of how a system matches its users' expectations - see for example[1]. The trouble with this view is that the expectations themselves can be mistaken and can change almost arbitrarily, based perhaps on experience with the system. In this paper a somewhat narrower interpretation of system reliability is taken, more in line with typical formal, and often quantitative, assessments of hardware reliability. Thus we regard system reliability as being related to the success with which a system provides the service specified. By this means the concept of the reliability of a system is separated from that of the reliance placed on it.

The history of the development of computing systems has seen some fascinating interplay between reliance and reliability. The reliability of early computers caused relatively little reliance to be placed on the validity of their outputs, at least until appropriate checks had been performed. Even less reliance was placed on the continuity of their operation - lengthy and frequent periods of downtime were expected and tolerated. As reliability increased so did reliance, sometimes in fact outdistancing reliability so that additional efforts had to be made to reach previously unattained reliability levels. During this time computing systems were growing in size and functional capacity so that, although component reliability was being improved, the very complexity of systems was becoming a possible cause of unreliability, as well as a cause of misunderstandings between users and designers about system specifications.

The informal but, it is hoped, rigorous definitions presented below of concepts relating to system reliability presume the existence of some external specification of the requirements that the system is supposed to meet. Ideally this specification will have

previously been agreed and documented; in practice, some aspects of it may exist only in the minds of persons authorised to decide upon the acceptability of the behaviour of the system. The terminology adopted here is defined in general terms, and is intended to relate to both hardware and software since in a complex computing system the reliability of both will be of great relevance to the overall reliability of the system.

2. SYSTEMS AND THEIR FAILURES

A system is defined to be a set of components together with their interrelationships, where the system has been designed to provide a specified service. The components of the system can themselves be systems and their interrelationship is termed the algorithm of the system. There is no requirement that a component provide service to a single system; it may be a component of several distinct systems. However, the algorithm of the system is specific to each individual system. The algorithm, plus the identification of the components, are sometimes termed the structure of the system.

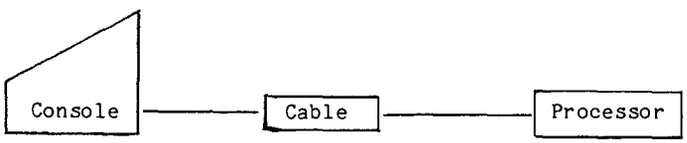


Figure 1. A Three Component System

Example

Figure 1 is a simple schematic representation of a system consisting of a processor, a console and an interconnecting cable - these three components are interrelated by being plugged together. The interconnecting lines in the diagram represent these relationships, rather than any physical component.

The reliability of a system is taken to be a measure of the success with which the system conforms to some authoritative specification of its behaviour. Without such a specification nothing can be said about the reliability of the system. When the behaviour of a system deviates from the specification, this is called a failure. Measures of the reliability of a system, such as Mean Time Between Failures (MTBF) and Mean Time To Repair (MTTR), can be based on the actual or predicted incidence of failures and their consequences.

Quantitative system reliability measures all concern the success with which a system provides its specified service. The much broader notion of reliability, which is here termed "reliance", relates to the situation where there is a lack of understanding and/or agreement as to the specification of a system, or where it is wished to differentiate between the relative acceptability of different kinds of (specified) system behaviour.

An internal state of a system is said to be an erroneous state when that state is such that there exist circumstances (within the specification of the use of the system) in which further processing, by the normal algorithms of the system, will lead to a failure which is not attributed to a subsequent fault.

The term error is used to designate that part of the state which is "incorrect". An error is thus an item of information, and the terms error, error detection and error recovery are used as casual equivalents for erroneous state, erroneous state detection and erroneous state recovery.

A fault is the adjudged cause of an error, while a potential fault is a construction within a system such that (under some circumstances within the specification of the use of the system) that construction will cause the system to reach an erroneous state. Such a fault may be classified as a mechanical fault or as an algorithmic fault, that is, a mistake in the algorithm of the system. Mechanical faults are those faults which are ascribed to the behaviour of a component of the system, that is, are failures of system components.

Example

A storage module which fails to store data correctly could be the fault which causes errors in the internal tables used by an operating system, errors which lead to a complete system failure, perhaps by causing the system to go into an unending loop.

Hopefully it will now be clear that the generality of the definitions of failure and fault has the intended effect that the notion of fault encompasses design inadequacies as well as, say, hardware component failure due to ageing. For example, it covers a mistaken choice of component, a misunderstood or inadequate representation of the specification (of either the component, or the service required from the system) or an incorrect interrelationship amongst components (such as a wrong or missing interconnection in the case of hardware systems or a program bug in software systems).

As the definition given above implies, identification of a state as erroneous involves judgement as to which fault is the cause of a particular failure. A demonstration that further processing can lead to a failure of a system indicates the presence of an error, but does not suffice to locate a specific item of information as the error. It can, in fact, be very difficult to attribute a given failure to a specific fault. Even precise location of an error is by no means guaranteed to identify a fault to which the failure may be attributed. Consider, for example, a system affected by an algorithmic fault. The sequence of internal states adopted by this system will diverge from that of the "correct" system at some point in time, the algorithmic fault being the cause of this transition into an erroneous state. But there need be no unique correct algorithm. It may be that any one of several changes to the algorithm of the system could have precluded the failure. A subjective judgement as to which of these algorithms is the intended algorithm determines the fault, the items of information in error, and the moment at which the state becomes erroneous. Of course, some such judgements may be more useful than others.

Example

Consider a program which is supposed to set Y and X to the initial value of X and the sum-check of the ten element vector A respectively. A faulty version of this might be

```
Y:=X; for i:=1 step 1 until 10 do X:=X+A[i];
```

which could be corrected either by inserting "X:=0;" just before the for statement, or "X:=X-Y;" after it. These two alternative corrections imply different judgements as to the exact fault, and as to which states of the computation evoked by the program were erroneous.

The significance of the distinction between faults and errors is most clearly seen when the repair of a system is considered. For example, in a data base system repair of a fault may consist of the replacement of a failing program (or hardware) component by a correctly functioning one. Repair of an error requires that the information in the data base be changed from its currently erroneous state to a state which will permit the correct operation of the system. In most systems, recovery from errors is required, but repair of the faults which cause these errors, although very desirable, is not necessarily essential for continued operation.

3. FAULT AVOIDANCE AND FAULT TOLERANCE

The traditional approach to achieving reliable computing systems has been largely based on fault avoidance (termed fault intolerance by Avizienis). Quoting Avizienis:[2] "The procedures which have led to the attainment of reliable systems using this approach are: acquisition of the most reliable components within the given cost and performance constraints; use of thoroughly refined techniques for the interconnection of components and assembly of subsystems; packaging of the hardware to screen out expected forms of interference; and carrying out of comprehensive testing to eliminate hardware and software design faults. Once the design has been completed, a quantitative prediction of system reliability is made using known or predicted failure rates for the components and interconnections. In a "purely" fault intolerant (i.e. nonredundant) design, the probability of fault-free hardware operation is equated to the probability of correct program execution. Such a design is characterised by the decision to invest all the reliability resources into high-reliability components and refinement of assembly, packaging and testing techniques. Occasional system failures are accepted as a necessary evil, and manual maintenance is provided for their correction."

There are a number of situations in which the fault avoidance approach clearly does not suffice. These include those situations where faults are likely to slip through into the system and it is inaccessible to manual maintenance and repair activities, or where the frequency and duration of the periods of time when the system is under repair are unacceptable. An alternative approach to fault avoidance is that of fault tolerance, an approach at present largely confined to hardware systems, which involves the use of protective redundancy. A system can be designed to be fault tolerant by incorporating into it additional components and abnormal algorithms which attempt to ensure that occurrences of erroneous states do not result in later system failures. The degree of fault tolerance (or "coverage") will depend on the success with which erroneous states are identified and detected, and with which such states are repaired and replaced.

There are many different degrees of fault tolerance which can be attempted. For example, a system designer might wish to reduce the incidence of failures during periods of scheduled operation by designing the system so that it will remain operational even in the presence of, say, a single fault. Alternatively, he might wish to attain very lengthy continuous periods of failure-free operation by designing the system so that it can tolerate not just the presence of a fault, but also the activity involved in repairing the fault.

Fault tolerant systems differ with respect to their behaviour in the presence of a fault. In some cases the aim is to continue to provide the full performance and functional capabilities of the system. In other cases only degraded performance or reduced functional capabilities are provided until the fault is removed - such systems are sometimes described as having a "fail-soft" capability.

Example

It is now typical for the computer terminals used in banks to incorporate significant processing and storage facilities. Such terminals enable data input and possibly some limited forms of data validation to continue even when the main computer system is not operational.

Schemes for fault tolerance also differ with regard to the types of fault which are to be tolerated. The design of fault-tolerant hardware systems is based on careful enumeration of expected faults due to component ageing, electrical interference and the like, and on complete identification of their consequences. Such systems have achieved considerable success. However, in general no attempt is made in such systems to cope with algorithmic faults in the hardware design, or in the associated software. Rather it is assumed that such faults have been successfully avoided and are not present in the system. This illustrates that fault tolerance and fault avoidance are better regarded as complementary rather than as competitive approaches to system reliability.

All fault tolerance measures depend on the effective utilisation of supplementary elements of the system which may be termed protective redundancy. The techniques which utilise this redundancy can be classified in various different ways; the general classification adopted here identifies strategies for (i) error detection; (ii) damage assessment; (iii) error recovery; and (iv) fault treatment.

The implementation of these strategies can take many different forms, as is discussed subsequently. In a given system the particular strategies used may vary in different parts of that system, and at different times during its operation. Indeed, it is not always possible to make a positive identification of the elements responsible for each of the constituent strategies used in a given fault tolerance technique, for, while the starting point is always the detection of an erroneous state, the order in which the other

strategies are carried out can vary, and there can be much interaction between them.

3.1. Error Detection

The starting point for all fault tolerance techniques is the detection of an erroneous state which could have led to system failure. Ideally, the checks performed by the error detection mechanisms should satisfy three criteria: firstly, they should be based solely on the specification of the service that the system is supposed to provide; secondly, they should check for absolute correctness of the behaviour of the system with respect to that specification; and thirdly, they should be independent from the system itself. In practice, of course, such rigorous checking cannot be attained. Therefore it is usual to attempt to enforce acceptability, a lower standard of behaviour than absolute correctness, with the hope that such checks will still enable a high percentage of errors to be detected. Acceptability checks can either be checks that the operation of the system appears to be satisfactory, or checks that specific erroneous situations have not arisen. A disadvantage of the latter approach is that only anticipated erroneous situations can be detected. However, for such situations checks can often be designed which are simpler and more specific than a general check that the operation of the system is satisfactory.

Some of the possible forms of checks in a system are: (i) replication checks, (ii) reversal checks, (iii) coding checks, (iv) interface checks, and (v) diagnostic checks.

Replication checks are a common form of error detection mechanism, involving replication of some part of the activity of the system to enable the consistency of the results to be checked. The type of replication used will depend on the type of faults that are anticipated (and of course on cost/performance constraints). Thus replication involving two or more components of identical specification but independent design would be employed if design faults were expected. Alternatively, replication involving two or more components of the same design or repeated use of the same component would be used to detect permanent or transient component faults respectively.

Reversal checks involve the processing of the results of a system to calculate what the input to the system should have been. The calculated input is then compared with that which actually occurred. Only certain types of system, where the inverse computation is relatively straightforward, lend themselves to this type of check.

Example

The JPL-STAR computer[3] employed "inverse microprogramming" which deduced what an operation should have been from the active gating signals. The deduced operation could then be compared with that actually requested.

Coding checks are also a common form of error detection, and often provide a means of reducing the cost of the error detection. Techniques such as parity, Hamming codes and cyclic redundancy codes use redundancy to enable the acceptability of a (possibly large) set of data to be checked. Checking large and complex masses of data is often infeasible without the use of coding techniques. However, it is at best a limited form of check, based on assumptions about the types and consequences of faults which might occur.

Example

Parity checks are regarded as being suitable for core stores, but not for telecommunications where the faults give rise to entirely different error characteristics.

All of the above forms of check will be founded on faith in the actual structuring of the system, based on the presumed effectiveness of the constraints that were applied to the interfaces of the system. Interface checks are one means of providing constraints, where mechanisms within components serve to check the interactions across interfaces. Checks for illegal instructions, illegal operands and protection violations are common examples of interface checks provided by hardware systems.

Diagnostic checks involves using a component with a set of inputs for which the expected outputs are known and can be compared with those actually obtained from the component. Diagnostic checks are usually used periodically, interspersed with periods of time during which it is assumed that the component is working correctly. The effectiveness of such checks will depend on the frequency of their use (with respect to the frequency of the occurrence of faults) and on the amount of time and resources that can be allocated to the checks. The problem with such checks is the amount of time for which errors could go undetected and spread throughout the system. Thus diagnostic checks are not often used as a primary error detection mechanism, but are used to supplement other mechanisms - for example for purposes of fault location.

3.2. Error Recovery

Recovery from the consequences of a fault involves transforming an erroneous state into a valid state from which the system can continue to provide the specified service. Two strategies for error recovery can be identified: error recovery provided to a system and involving the restoration of a prior state of the system (or part of the system) is termed backward error recovery. In contrast, forward error recovery involves the system itself making further use of its present erroneous state to obtain another state. The aim of both strategies is to attain a state which it is hoped is free from errors.

Backward error recovery involves the provision of recovery points, that is points in time at which the state of the system is (at least conceptually) saved for future reinstatement if required. Various techniques can be used for obtaining such recovery points. Checkpointing-type mechanisms involve foreknowledge of the resources that the processes could modify (e.g. all of working storage). Audit trail techniques involve recording all the modifications that are actually made. Recovery cache-type mechanisms are a compromise which involve recording the original states of just those resources which are modified (see below). Because of the cost of maintaining a large number of recovery points, they are usually explicitly discarded when it is hoped they are no longer required (e.g. re-use of a back-up tape). The implied reduction in recovery capability which the loss of a recovery point entails is called commitment.

The major importance of backward error recovery is that it is a simple technique which makes no assumptions about the nature of the fault involved (apart from assuming that the fault has not compromised the recovery mechanism), and that no effort need be expended on damage assessment. Thus, if it is available, backward error recovery is a general recovery mechanism and can provide recovery after all types of faults, even unanticipated faults in the design of the system. However, there may be situations in which it is an expensive recovery mechanism in that it involves undoing all of the activity of the system since the recovery point was established, not just those parts which were erroneous.

As described above, forward error recovery involves the system itself transforming its erroneous state into a valid state. However, the problem with forward error recovery techniques is that they must rely heavily on the knowledge of the nature of the fault involved and its exact consequences (influenced by knowledge of the structure of the sys-

tem) and in consequence they are inseparable from the problems of damage assessment and of providing a continued service. Thus forward error recovery has to be designed specifically for each system. Nevertheless, in situations where a fault and its full consequences can be anticipated, forward error recovery can provide efficient and simple recovery.

3.3. Recovery Blocks

The recovery block scheme has been introduced as a means of providing fault tolerance in programs. It is intended for coping with faults whose exact location and consequences have not been (and, as with design faults, cannot be) predicted. It therefore is based on the use of backward error recovery.

The scheme 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.

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). 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 Figure 2, double vertical lines define the the extents of recovery blocks, while single vertical lines define the extents of alternate blocks, primary or otherwise.

```
A: ensure AT
    by      AP : begin
                     <program text>
                 end
    else by AQ : begin
                     <program text>
                 end
    else error
```

Figure 2: Simple Recovery Block.

```
A: ensure AT
   by        AP : begin   declare Y
                          <program text>
                    B : ensure BT
                        by        BP : begin declare U
                                                <program text>
                                       end
                        else by   BQ : begin declare V
                                                <program text>
                                       end
                        else by   BR : begin declare W
                                            <program text>
                                       end
                        else error
                        <program text>
                 end
   else by   AQ : begin   declare Z
                          <program text>
                    C : ensure CT
                        by       CP : begin
                                         <program text>
                                      end
                        else by  CQ : begin
                                         <program text>
                                      end
                        else error
                    D : ensure DT
                        by       DP : begin
                                      <program text>
                                      end
                        else error
                 end
   else error
```

Figure 3: A More Complex Recovery Block.

Figure 3 shows that the alternate blocks can contain, nested within themselves, further recovery blocks. In this figure 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.

3.3.1. Acceptance Tests

The function of the acceptance test is to try and ensure that the operation performed by the recovery block is to the satisfaction of the program which invoked the block. It supplements any checking performed within the block by more conventional means, such as run-time assertions. 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 ideally the acceptance test should 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 rigour 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.

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.

When an acceptance test is being evaluated, any non-local 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 rigour, the acceptance test is able to access such variables either for their modified value or for their original (prior) value.

```
ensure sorted (S) ? (sum(S) = sum(prior S)
by quickersort (S)
else by quicksort (S)
else by bubblesort (S)
else error
```

Figure 4: Fault-Tolerant Sort Program

Figure 4 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.

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

```
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
```

Figure 5: Recovery Block with Alternates which achieve different, but still acceptable though less desirable, results.

Figure 5 shows a recovery block consisting of a variety of alternates. (This figure is taken from [4].) 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.

3.3.3. 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. In particular the error-prone task of explicit preservation of restart information is avoided.

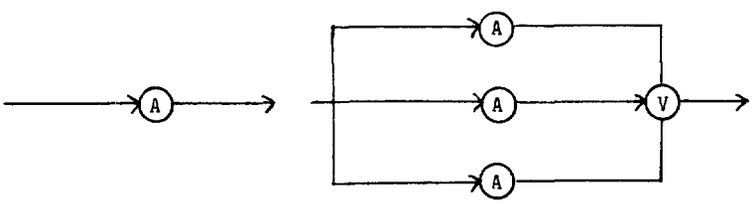
Whenever a process has to be backed up, it is to the state it had reached just before entry to the primary alternate - therefore the only values that have to be reset are those of nonlocal variables that have been modified. Since no explicit restart information is given, it is not known beforehand which 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 "recovery 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 recognising 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 recovery 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. We have in fact constructed one hardware implementation, in the form of a device which sits astride a PDP-11 Unibus, and monitors store operations issued by the processor[6]. 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 generalisation of the facility in CAP's "middleware" scheme[7] for preventing individual application programs from destructively updating files. As mentioned earlier, the recovery block scheme is effectively a form of stand-by sparing. Another well established approach to conventional fault tolerance is Triple Modular Redundancy.

In its standard application, TMR is used to provide tolerance against hardware component failures. Thus, to tolerate the failure of component A in Figure 6a, it could be replaced by the TMR system in Figure 6b, consisting of three copies of component A (each of identical design) and majority voting circuits V which check the outputs from these components for equality. The system in Figure 6b is therefore designed to tolerate the failure of any single A component by accepting any output on which at least two components agree.



(a) (b)

Figure 6: Triple Modular Redundancy.

Clearly, the standard TMR structure provides no protection against design faults. However one could envisage a TMR-like structure involving modules of "independent" design. This approach has been investigated under the title N-version programming[8]. In those situations where one can reasonably hope to achieve a large measure of independence of design, yet nevertheless require identicality of outputs from each module, such a scheme

has a certain attractiveness, particularly if the processing resources allow parallel execution of the modules. In general however the requirement for identicality is overly stringent, and so one is led into complicating the voters, so as to allow "sufficiently similar" answers to be regarded as equivalent. Moreover, as with conventional TMR, this approach only addresses the problem of masking isolated faults, and does not cope with multiple faults, or of faults involving interactions with the surrounding environment. However it is to be preferred, as a means of error detection, over an acceptance test whose complexity is so great as to rival that of the alternates that are being checked.

4. ANTICIPATED VERSUS UNANTICIPATED FAULTS

The specified service that a component of a system is designed to provide might include activities of widely differing value to its environments. No matter how undesirable, none that fall within the specifications will be termed failures. However the specification can be structured so as to differentiate between a standard service, and zero or more exceptional services. For example, the standard service to be provided by an adder would be to return the sum of its inputs, exceptional services to indicate that an arithmetic overflow has occurred, or that an input had incorrect parity.

Within a system, a particular exception is said to have occurred when a component explicitly provides the corresponding exceptional service. The algorithm of the system can be made to reflect these potential occurrences by incorporating exception handlers for each exception.

These definitions match the intent, but not the form of the definitions given by Goodenough[9], who states:

"Of the conditions detected while attempting to perform some operation, exception conditions are those brought to the attention of the operation's invoker ... In essence, exceptions permit the user of an operation to extend an operation's domain (the set of inputs for which effects are defined) or its range (the effects obtained when certain inputs are processed)."

However, in contrast to Goodenough, we have taken care to avoid the use of the word 'failure' in discussing exceptions. This is not mere pendantry. Rather it is a consequence of the very basic view we take of failures, namely as occurring when and only when a system or component does not perform as specified. Although a system designer might

choose to treat certain exceptions as component failures (which he might or might not provide abnormal algorithms to deal with), we regard the various schemes for exception handling (e.g. Parnas[10], Goodenough[9] and Wasserman)[11] and our technique of recovery blocks as complementary rather than competitive.

A basic feature of the recovery block scheme is that, because no attempt is made to diagnose the particular fault that caused an error, or to assess the extent of any other damage the fault may have caused, recovery actions have to start by returning the system to a prior state, which it is hoped precedes the introduction of the error, before calling an alternate. Should this prior state not precede the introduction of the error, more global error detection, and more drastic error recovery, is likely to occur later.

When exceptions are treated as component failures in a software system that uses recovery blocks, they will lead to the system being backed up to a prior state and an alternate being called. This will be appropriate when the exception is undesirable, and the system designer does not wish to provide an individual means of dealing with it.

Putting this the other way, exceptions can be introduced into the structure of a system which uses recovery blocks, in order to cause some of what would otherwise be regarded as component failures (leading to automatic back-up) to be treated as part of the normal algorithm of the system, by whatever explicit mechanisms the designer wishes to introduce for this purpose. Failures might of course still occur, in either the main part of the algorithm, or in any of the exception handlers, and if they do they will lead to automatic back-up. Such introduction of exceptions can therefore be thought of as a way of dealing with special or frequently occurring types of failure, in the knowledge that the recovery block structure remains available as a "back-stop".

However we would argue strongly against relying on exception handling as a means of dealing with algorithmic faults. Programmed exception handling involves predicting faults and their consequences, and providing pre-designed means of on-line fault diagnosis. Thus although it can be of value in dealing with foreseen undesirable behaviour by hardware components, users, operating staff, etc., it is surely not appropriate for dealing with software faults - predictable software faults should be removed rather than tolerated. Indeed the incorporation of programmed exception handlers to deal with likely software faults would in all probability, because of the extra complexity it would add to the software, be the cause of introducing further faults, rather than a means of coping with

those that already exist. On the other hand when used appropriately for anticipated faults of other types they can provide a useful means of simplifying the overall structure of the software, and hence contribute to reducing the incidence of residual design faults.

```
1  . . .
2  ensure consistent_inventory by
3  process_updates: begin integer num;
4                         exception goof = overflow or underflow or conversion;
5                         procedure checknum (integer j);
6                                 global integer count = 0;
7                                 procedure message;
8                                       begin count := count +1;
9                                             write ("please try again");
10                                            if count ? 3 then
11                                               begin write ("three strikes -
                                                                you're out);
12                                                     signal error
13                                               end
14                                               else retry;
15                                      end message;
16                                begin/* body of checknum */
17                                      . . .
18                                      read(j) [goof : message, ioerr:error]
19                                end checknum;
20                        begin/* start of main body */
21                             . . .
22                             while updates_remain do
23                                   begin update_no := update_no +1;
24                                        . . .
25                                        checknum(num);
26                                        . . .
27                                   end
28                             . . .
29                        end main body
30                 end process_updates
31 else by
32 refuse_updates : begin write ("sorry - last update accepted was number");
33                        write (update_no)
34                 end
35 else error
36 . . .
```

Figure 7: An Example of a Program which Incorporates Programmed Exception Handling within a Recovery Block Structure.

Figure 7 shows a section of program text which incorporates programmed exception handling within a recovery block structure. The example, and the form of exception handling shown, are based on that given by Wasserman[11].

The basic form of the example is

```
ensure consistent_inventory
by process_updates
else by refuse_updates
else error
```

The implicit assumption is that the program is maintaining an inventory file whose consistency is to be checked after each related sequence of updates, to determine whether this sequence can be incorporated. The updating process uses the procedure 'checknum' to read and check the updates. This procedure provides an exception handler for some of the exceptions that can be raised by the 'read' routine, so that the person providing the inputs can have two chances of correcting each input.

The procedure 'checknum' is taken directly from Wasserman[11], but has been simplified to take account of error recovery facilities provided by the recovery block structure in which it is used. More detailed notes on the example follow.

Line 2 The Boolean expression 'consistent_inventory' will be evaluated if and when 'process_updates' reaches its final 'end'. If the expression is true, the alternate 'refuse_updates' will be ignored and the information stored by the underlying recovery cache mechanism, in case the effects of 'process_updates' had to be undone, will be discarded. Otherwise this information will be used to nullify these effects, before 'refuse_updates' is called, after which the Boolean expression 'consistent_inventory' is checked again.

Line 4 In Wasserman's scheme a group of separate exceptions can be gathered together, as here to define the exception using the exceptions 'overflow', It is assumed that all three can be signalled by the routine 'read' - the first two perhaps being built-in exceptions that the hardware signals, the third being implemented by the routine 'read' itself.

Line 7 The procedure 'message' is an exception handler defined within 'checknum'. The first two occasions on which it is called it uses Wasserman's scheme for retrying the procedure which raised the exception (see line 14), but on the next occasion it signals error.

Line 18 Here 'checknum' calls 'read' and arranges that the exception 'goof' (i.e. the exceptions 'overflow', 'underflow' or 'conversion') will be handled by the procedure 'message', but that if 'read' signals 'ioerror' this will cause 'process_updates' to be abandoned.

Line 20 All that is illustrated of the main body of 'process_updates' is that it counts the number of updates, which it reads and checks using the routine 'checknum'.

Line 32 The second alternate 'refuse_updates' is called if the first alternate 'process_updates' abandons its task, or fails to pass the acceptance test, for any reason (including of course, any residual design error within its code). If this happens, all changes that 'process_update' has made to the inventory will be undone, and the integer 'update_no' will be reset. This integer is then used for an apologetic message to the user.

5. LEVELS OF ABSTRACTION

In choosing to regard a system (or its activity) as made up of certain components and to concentrate on their interrelationships whilst ignoring their inner details, the designer is deliberately considering just a particular abstraction of the total system. When further details of a system (or part of a system) need to be considered, this involves a lower level of abstraction which shows how a set of interrelated components are implemented and act, in terms of some more detailed components and interrelationships (which will of course in turn just be abstractions of yet more detailed components and interrelationships, and so on).

The identification of a set of levels of abstraction (each of which might relate to the whole system, or just some part of the system) and the definition of their interrelationships again involves imposing a structure on a system, which is referred to here as vertical structuring. Thus vertical structurings describe how components are constructed, whereas horizontal structurings describe how components interact.

The importance of levels of abstraction is that they allow the designer to cope with the combinatorial complexity that would otherwise be involved in a system constructed from a very large number of very basic components. The price that is paid is the requirement for well-documented specifications of the external characteristics of each level - such specifications can be thought of as the abstraction interfaces interposed between levels, much as the specifications of the interrelationships between interacting components within a level could be termed communication interfaces. In each case the interface will, if well chosen, allow the designer to ignore (at least to some extent) the workings of those parts of the system which lie on the far side of the interface.

An abstraction interface may be conveniently thought of as being characterised by a language providing objects and their associated operations. Such interfaces can, for example, be implemented using programming languages which allow the introduction of new

abstract types.

Abstraction interfaces can in themselves contribute to the overall reliability of a system, by simplifying the tasks of its designer. Moreover they are a means by which some reliability problems can be conveniently, and more-or-less completely, hidden from the designers of other parts of a system. For example a real (i.e. fallible) magnetic tape drive might, by use of a device handler which does automatic retries, be used as the basis of an abstract type (e.g. an I/O stream) which users assume to be totally reliable. On a grander scale, an operating system nucleus might be used to provide automatic hardware reconfiguration so as to avoid further use of failing units, and hide all such problems and activities from application programs. However abstraction interfaces can still be of use even when underlying faults cannot be (assumed to be) completely masked, providing of course that the faults are not such as to destroy the interface itself.

One possibility is for the interface to have been designed so as to permit the underlying level, after having detected an error and exhausting any error recovery capability it might possess, to report the error to the level above as an exception. Thus the level above, though being provided with more complicated objects than its designer might have wished (e.g. a file store whose 'write' operations can result in a 'disk full' exception) can be structured so as to delegate responsibility for such situations to exception handling routines. Alternatively, backward error recovery might be invoked in the level above, just as it might be if this level detected an error for itself for which it had no forward error recovery provisions.

Backward error recovery necessitates that any abstract objects which are used to represent the state of a process be themselves 'recoverable'. Such objects could of course simply be constructed out of smaller recoverable objects (e.g. recoverable complex variables could be constructed from pairs of words held in a memory equipped with a recovery cache). Alternatively, responsibility for the provision of recoverability could be vested in the programs that implement the corresponding abstract type.

One approach to such provision is illustrated in the abstract type definition given in Figure 8, where it will be seen that three extra operations are provided just for purposes of recoverability in a process using this type. The first of these extra operations, Establish Recovery Point (erp), will be called automatically each time the process in which this type definition is embedded enters a further recovery block. The second

operation Reverse will be called if the process has to return to its last recovery point. The final operation Accept will be called if the process passes an acceptance test so that a recovery point can be discarded.

```
type recoverable file;
begin
   write: <store in appropriate disk location>
   read: <fetch from disk>
   (erp: <arrange to use new area of disk for writing>
   reverse: <discard disk areas written since last 'erp'>
   accept: <discard prior versions of
                              blocks written since last 'erp'>)
end
```

Figure 8: Definition of a recoverable file

None of the three extra operations would be directly callable. Rather what is in effect a generalisation of the recovery cache is used to control their invocation. The cache will record descriptors for the Reverse and Accept operations corresponding to objects for which recovery information has been 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 Reverse operations 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 Accept operations.) The program of Reverse/Accept calls is initially null, but grows as the process performs actions which add to the task of backing it up. As with the basic recovery 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 is completed successfully, some entries will be discarded, but those that relate to objects which are nonlocal to the enclosing environment will be consolidated with the existing "program" in the underlying region of the cache.

6. RECOVERY IN CONCURRENT SYSTEMS

At a suitable level of abstraction, a computer system can be viewed as consisting of a family of concurrent processes that are harmoniously exchanging and processing the information entrusted to the system. These processes may be computational activities of the computer(s), the users of the system or a combination of both. Also, the information exchange may take the form of a message exchange between two or more computational activities of the computer(s); a conversation between a user (at a terminal) and the

computer(s); users of the system exchanging information about the system through the postal system; or any combination of these and various other ways of exchanging information. In order to discuss the recovery actions of such processes (for example, what actions to undertake should it be decided that the information sent by a process to other processes was incorrect) some form of model is needed which suitably abstracts away the differences outlined above.

The single most important concept in the study of multi-process systems is that of an atomic action. An atomic action is an indivisible unit of computation involving one or many processes which conceptually may be executed instantaneously, without interference from (= communication with) processes not participating in the action. The indivisibility of atomic actions is purely relative, depending on viewpoint. As illustrated in Figure 9, atomic actions may be nested inside other atomic actions at a different level.

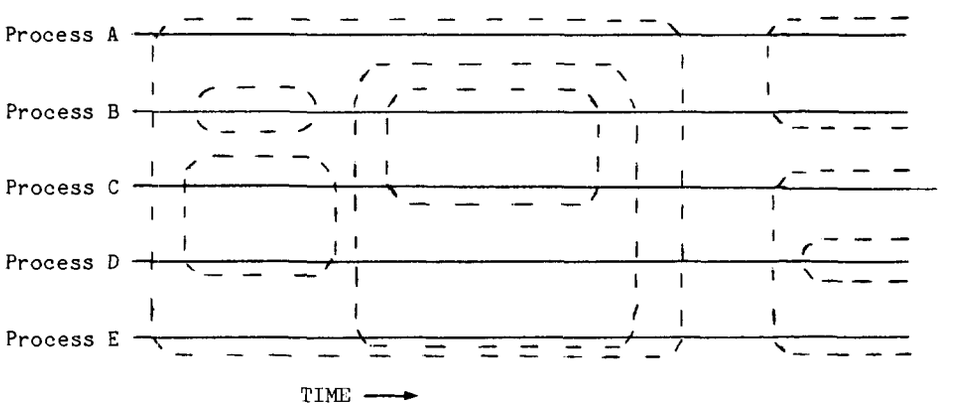


Figure 9: Atomic Actions at Different levels.

Much of the theory concerning serialisability of data-base transactions is related to the concept of atomic actions. Atomic actions are significant in the context of recovery control of multi-process systems by virtue of the fact that they can be made to define boundaries of the "area" of a computation which should be abandoned (for backward error recovery) or repaired (for forward error recovery) after an error has been detected. There are two fundamentally different approaches to recovery control in multi-process systems with respect to atomic actions: namely the planned and the unplanned strategies.

In the former approach, processes co-operate to participate in pre-arranged shared atomic actions, known as conversations, resulting in the system structure depicted in Figure 9 above. Each participating process establishes a recovery point on entry, and all

will be made to recover if any one of them detects an erroneous state while within the conversation.

In the latter approach, the processes make no attempt to set up atomic actions. Inter-process communication is random and no attempt is made to synchronise the establishing of recovery points. When recovery action is initiated by one process, the system endeavours to fit an atomic action (a "sphere of control") around the computation retroactively, based on the observed information flow between the processes, in order to determine to what consistent state the system should be recovered. The lack of pre-planned atomic actions may allow information flow to circumvent recovery points so that 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.

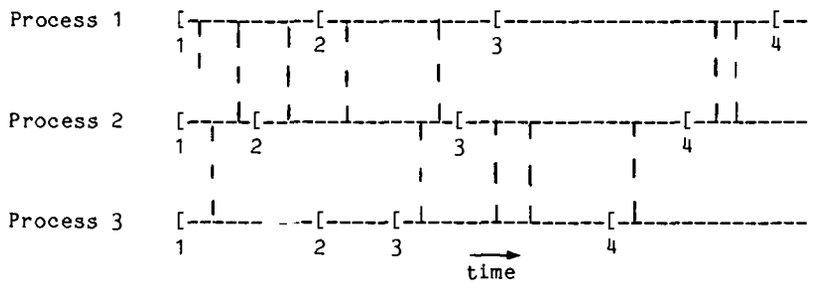


Figure 10: The Domino Effect.

The problem is illustrated in Figure 10, 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 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!

7. DISTRIBUTED SYSTEMS

The problem of providing system-wide fault tolerance in a multi-process system is exacerbated in loosely coupled distributed systems with decentralised control. There is no single entity maintaining a global view of the system, and so the processes have to

co-operate to exchange control information as well as data. Moreover, the absence of any centralised control in a system poses new recovery problems. For example, parts of a system may fail while the remainder of the system continues processing having received no notification of the failure; this may further compromise the integrity of the system.

Fortunately, it is possible to impose a structure on distributed systems so that the complexities resulting from decentralised control become manageable. To show how this may be done, it will be assumed that a distributed system consists of a number of autonomous processing nodes connected together by communicating links. Execution of a user job in such a system consists of concurrently running parts of the job at the appropriate nodes. Traditionally, a user job is executed by a single process in a centralised system. The same concept can be extended to apply to distributed systems such that there is a process at each of the appropriate nodes to execute the relevant parts of the job.

The activities of such a group of processes (executing a given job) can be co-ordinated so that at a higher level of abstraction, the group appears as a single process. Such a group of processes has been termed a cohort by Gray[12]. Naturally, it is necessary for the operating systems of the various nodes to communicate with each other in order to provide the abstraction of cohorts.

Two ingenious (and closely similar) algorithms have been suggested recently for a cohort to be able to abandon, when necessary, what at the level of the user program is seen as a transaction, i.e. a simple recoverable atomic action[13,12]. The key problem is that of providing means for the members of the cohort to know when it is safe for them to commit, and discard their individual recovery points. This is solved by treating one member of the cohort as a controller - the decision to commit is thus centralised. The controller first allocates the sub-tasks each member should carry out, and waits for their completion. If the controller does not receive "done" messages from all of the members, it decides to abort that transaction and sends "abort" messages to the members. On the other hand, if the controller receives "done" messages from all of its members, it decides to commit the transaction and sends "commit" messages to all of its members; the controller makes sure that all of the members receive the "commit" message. This behaviour gives the algorithm its name, that is the 'two-phase commit' protocol.

Unplanned recovery control in distributed systems involves not just tracing back, via records of information flow, but also perhaps having recovery control messages 'chase' the

continued flow of data between processes. During this time further faults might occur and further errors might be detected. Moreover, individual processes might unilaterally discard recovery points. The so-called "chase protocol" has been developed to solve these problems. Somewhat surprisingly it does not consist of separate phases of halting further progress, causing individual processes to go back to appropriate recovery points, and then restarting the processes. Rather these various activities are all meshed together - a full description is given in [14]. A possible alternative to either planned or unplanned recovery control for backward error recovery is to use forward error recovery instead, in the form of 'compensation'. This involves the process that has belatedly detected that it has sent wrong information to another, providing supplementary information intended to correct the effects of information that it had previously sent. This requires that both (or more generally, all) the interacting processes are designed such that, when erroneous information is found to have been sent out or let out by a process, all of the other processes are capable of accepting the corrective information which is then sent out by, or on behalf of, the offending process.

Example

If a stock control data base has been updated because of an input message indicating the issuance of some stocked items, and it is later found that this message was incorrect, it might be possible to compensate for the wrong message by means of another message purporting to record the acquisition of replacement stock items. Such a simple compensation would probably not suffice if, for example, the stock control system had, as a result of the wrong information, recalculated optimum stock levels, or produced purchase orders for the replenishment of the stock involved.

Presumably the atomic action concept can be used as a basis for automating the task of determining when and where pre-specified compensation algorithms are to be applied. To date however such compensation as is attempted is usually done manually by the users of a system, who can expect precious little help from the system in carrying out the task. Thus when the users of a data base system for example, belatedly recognise that it has been given and has accepted incorrect information, the task of finding a set of updates which can bring the data base back into correspondence with the outside reality it is intended to model can be very difficult indeed.

8. CONCLUSIONS

The subject of fault-tolerant computing design has been developed over a number of years, though to date most of the concentration has been on the provision of protective redundancy in hardware. These notes have instead dealt mainly with overall system fault tolerance, and in particular with the role that software can play, both as a source of unreliability, and as the means of implementing fault tolerance stratagems. Even so, many topics have been treated only cursorily, and readers are referred to[15] for further details, and an extensive bibliography.

9. ACKNOWLEDGEMENTS

These notes are based in large part directly on earlier papers by various members of the Newcastle reliability project.

References

1. P. Naur, "Software Reliability," pp. 243-251 in State of the Art Report on Reliable Software, Infotech, London (1977).

2. A. Avizienis, "Fault-Tolerant Systems," IEEE Transactions on Computers Vol. C-25(12), pp.1304-1312 (1976).

3. A. Avizienis et al, "The STAR (Self-Testing and Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design," IEEE Transactions on Computers Vol. C-20(11), pp.1312-1321 (November 1971).

4. T. Anderson and R. Kerr, "Recovery Blocks in Action: A System Supporting High Reliability," Proceedings of 2nd International Conference On Software Engineering, pp.447-457 (October 1976).

5. J.J. Horning, H.C. Lauer, P.M. Melliar-Smith, and B. Randell, "A Program Structure for Error Detection and Recovery," pp. 171-187 in Lecture Notes in Computer Science 16, ed. E. Gelenbe and C. Kaiser, Springer Verlag (1974).

6. P.A. Lee, N. Ghani, and K. Heron, "A Recovery Cache for the PDP-11," Digest of Papers FTCS-9, pp.3-8 (June 1979). (Also TR134, Computing Laboratory, University of Newcastle upon Tyne)

7. B. Randell, "Highly Reliable Computing Systems," TR20, Computing Laboratory, University of Newcastle upon Tyne (1971).

8. L. Chen and A. Avizienis, "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," Digest of Papers FTCS-8, pp.3-9 (June 1978).

9. J. B. Goodenough, "Exception Handling: Issues and a Proposed Notation," Communications of the ACM Vol. 18(12), pp.683-696 (December 1975).

10. D.L. Parnas , Response to Detected Errors in Well-Structured Programs, Dept. of Computer Science, Carnegie-Mellon University, Pittsburg (July 1972).

11. A.I. Wasserman, Procedure-Oriented Exception Handling, Medical Information Science, University of California, San Francisco (1976).

12. J.N. Gray, "Notes on Data Base Operating Systems," pp. 393-481 in Lecture Notes in Computer Science 60, ed. R. Bayer, R.M. Graham and G. Seegmueller, Springer-Verlag, Berlin (1978).

13. B.W. Lampson and H.E. Sturgis, "Crash Recovery in a Distributed Data Storage System," Xerox PARC Report (April 1979).

14. P.M. Merlin and B. Randell, "Consistent State Restoration in Distributed Systems," Digest of Papers FTCS-8, pp.129-134 (June 1978). (Also TR113, Computing Laboratory, University of Newcastle upon Tyne)

15. T. Anderson and B. Randell (eds.), Computing Systems Reliability, Cambridge University Press, Cambridge (1979).