A reconsideration of the recovery block scheme

P. A. Lee

Computing Laboratory, The University, Newcastle upon Tyne, NE1 7RU

The recovery block scheme has been introduced as a method of providing fault tolerance at the software level in a computing system. From the widespread interest that has been expressed in the scheme there appears to be a standard set of questions which are posed about its implementation and utility. This paper presents a brief overview of the recovery block scheme and then examines in detail the issues that these questions raise.

(Received January 1978)

The reliance which is being placed on present day computing systems has led to an increasing demand for reliability, particularly at the software levels in a system. Most techniques for producing reliable software (for instance, methodologies for program construction and testing) have concentrated on the praiseworthy aim of eliminating faults from the software before reliance is placed on its behaviour. However, it is widely recognised that complex systems are likely to contain residual design faults, both in the software and hardware. Efforts aimed at providing tolerance against such faults should have a beneficial effect on the reliability of a system. The recovery block scheme was introduced by Horning, *et al.* (1974) as a method of providing fault tolerance at the software levels in a system, particularly against residual design faults. The concepts of the recovery block scheme have been widely presented by various members of the Science Research Council sponsored 'Reliability Project' at Newcastle University, and from the questions asked at such presentations it would appear that there is a fairly standard set of doubts and misunderstandings about the scheme. The purpose of this paper is twofold: firstly, to present these questions with their answers, in the hope of clarifying the issues they raise; and secondly, to relate the recovery block scheme to some of the recent work of the Reliability Project.

The recovery block scheme

For completeness, this section presents a brief overview of the recovery block scheme. For further details the reader is referred to Horning, *et al.* (1974), Randell (1975) and Anderson and Kerr (1976).

Recovery blocks provide a means for a programmer to specify redundancy at the software level in a system by means of standby-spare algorithms which are used, as necessary, to replace failing algorithms. The outline of a recovery block is presented in **Fig. 1.**

The essential components of a recovery block are a set of algorithms (called *alternates*) and an *acceptance test*. The alternates are simply statement lists, with the first or primary alternate representing the preferred algorithm. The acceptance test is a programmer-provided error detection mechanism to check on the acceptability of the results produced by the

```
ensure ⟨acceptance test⟩
by ⟨first (primary) alternate⟩
else by ⟨second alternate⟩
. . .
. . .
else by ⟨nth alternate⟩
else error
```

Fig. 1 Recovery block outline

alternates. Although the recovery block scheme fits most easily into block structured languages such as ALGOL and PL/I, it can in fact be used with other high and low level languages. Indeed, the scheme can also be used for much more abstract levels in computing systems, such as those which support job control languages and data base accessing. However, this paper concentrates on its use within individual sequential programs.

The execution of a recovery block is as follows: on initial entry, the primary alternate is entered. At the end of the alternate the acceptance test (a boolean expression) is evaluated—if this test yields 'true' (that is, the results from the alternate are acceptable) then the recovery block is exited. However, if the acceptance test yields 'false', or if an error is detected by the underlying machine during the execution of the alternate, then *backward error recovery* occurs in that the state of the program is automatically reset to the state that existed when the recovery block was entered, and then the sequence of execution described above is repeated except that the next alternate is used in place of the failing alternate. A record of the errors and acceptance test failures which occurred is produced for subsequent use by the programmer. If all of the alternates fail, then this is regarded as a failure of that recovery block and an error condition is raised. As recovery blocks can be nested to any depth (conceptually at least) the failing recovery block may itself be embedded in an alternate of an enclosing recovery block. If this is the case then the error condition will result in the failure of the enclosing alternate. Otherwise, the program is terminated.

The underlying machine which is executing the programs containing recovery blocks provides the mechanisms for switching control between alternates and to enable the backward error recovery of the objects of the program to be accomplished. (Objects for which backward error recovery is provided will be termed *recoverable objects*.) These mechanisms will be transparent to the program and, for example, could be built into the hardware of the machine. A mechanism called the *recursive* or *recovery cache* has been proposed to implement the backward error recovery. The recovery cache essentially provides three functions: (*a*) recording recovery data; (*b*) performing recovery; and (*c*) discarding recovery data when recovery is no longer required. There are several ways in which the recording of recovery data can be implemented. The basis of the method proposed in the three papers referenced above is as follows: when an object which is not local to an alternate is updated for the first time from within that alternate, the original value of that object together with its address will be stored in the recovery cache, and will be used to restore the state of that object if recovery is invoked. Thus a minimum of recovery data is maintained in the recovery cache. (This method implemented in hardware would be appropriate for providing backward error recovery for simple objects such as integers, reals and characters, and structures of the same.) Other implementations for recording recovery data will be discussed subsequently. The underlying machine will also provide mechanisms to detect errors in the execution of programs. Typical errors detected would include illegal instructions, division by zero and memory access violations.

Doubts and misunderstandings

Q1. What types of fault are recovery blocks intended to provide tolerance against?

A1. There are essentially two types of fault that can occur in a system: (*a*) component faults, when a component does not function according to its specification; and (*b*) algorithmic faults, which are faults in the interrelationships between components. (This fault classification is discussed in more detail by Randell, Lee and Treleaven (1978).) At the hardware level in a system both component and algorithmic faults can occur, algorithmic faults being missing or incorrect connections between components. At the software level, faults are algorithmic (although can be regarded as component faults at another level of abstraction). Algorithmic faults are residual design faults in a system. The location and effects of such faults are unanticipatable, since in general they arise from unmastered complexity in the design of the system. Recovery blocks are designed to provide tolerance against algorithmic faults in both the software (program 'bugs') and in the hardware. After recovery, such faults are avoided by switching to the next alternate in the hope, perhaps vain, that the set of circumstances that led to the failure of the previous alternate are not repeated. Treatment of the fault is left for manual off-line diagnosis (aided by the error record mentioned previously) and repair.

A recovery block can also provide tolerance against some anticipated component faults in the underlying machine. If the underlying machine detects an error which it can attribute to a fault in its operation but which may have caused damage to the executing program, than an automatic *retry* facility can be provided—the error recovery for the program can be invoked to rectify any damage that may have been caused to the data of the program, and the same alternate re-entered. In this way, a recovery block can provide tolerance against the damage that has been caused either by transient component failures or by permanent component failures which result in reconfiguration or replacement of components at the hardware level.

Q2. As recovery blocks increase the size of the programming task, in that the alternates have to be programmed, surely the use of recovery blocks will increase the complexity of the program and therefore detract from, rather than increase, the overall reliability?

A2. It is true that the use of recovery blocks increases the size of the programming task. However, each alternate in a recovery block, when executed, starts from exactly the same state because of the error recovery capability that is provided. Thus the design of each alternate can (and preferably should) be independent of any other. The designer of one alternate need have no knowledge of the design of the other alternates, leave alone any responsibility for coping with any damage that a previous alternate might have caused. Equally, the designer of a program containing recovery blocks does not necessarily have to be concerned with which of the various alternates was eventually used. It is therefore argued that the increase of size in programs containing recovery blocks does not provide a corresponding increase in complexity. Indeed, the structure of recovery blocks may provide a means of reducing the complexity found in systems which have extensive ad hoc error detection and recovery facilities.

Q3. Is it always possible to generate alternative algorithms for a particular problem?

A3. The simple answer to this question is yes. The justification for this answer is as follows: there are essentially two different ways in which a recovery block can be used. The first and obvious situation is when it is required that each alternate of the recovery block produces exactly the same results. For some problems it is comparatively easy to obtain different algorithms for the alternates—sorting and mathematical functions such as integration are obvious examples. However, for other problems it may be difficult for a programmer to design different algorithms, particularly without making the same mistakes in each algorithm. To overcome this difficulty it is likely that separate programmers will be required, each working independently on a specification of the problem to provide an alternate to be incorporated in the final program. This is not a new concept and, for example, has been advocated by Gilb (1974) and Fischler, *et al.* (1975).

A further source of alternates may be obtained from previous versions of an algorithm. A common occurrence with software products is that a new version is introduced, often simply for performance considerations. Clearly, there will be situations in which the previous version can be used as a secondary alternate enabling the new version to be introduced into the system with the knowledge that if (when) it failed, then the original version was still available as a backup.

The second situation in which recovery blocks can be used is to provide what may be termed *graceful degradation in software*. It is not necessary that each alternate of a recovery block produces exactly the same results; the constraint on the alternates is that they produce acceptable results, as defined by the acceptance test. Thus, while the primary alternate attempts to produce the desired results, the second and subsequent alternates may only attempt to provide an increasingly degraded service. The more degraded the service, the simpler the alternate may be and consequently the greater the hope that it does not contain any design faults. Similarly, as each alternate is essentially different, it is more likely that a design fault will not be repeated in all alternates, whether produced by the same programmer or not. As an example of a recovery block designed in this manner, consider the part of a program that has to enter a disc-to-core transfer request into a queue of outstanding requests. The outline of such a program is presented in **Fig. 2.**

The acceptance test for this recovery block simply checks that the transfer queue is in a consistent state. The primary alternate attempts to place the new transfer request in the optimal position in the queue, for example, to minimise disc head movement. The second alternate avoids the complications of the primary alternate by simply placing the new request at the end of the queue. The third alternate is more desperate, and leaves the existing queue alone, providing a warning that the new request has been ignored. While this may cause

```
ensure ⟨consistency of disc transfer queue⟩
by ⟨algorithm which enters request
  in optimal queue position⟩
else by ⟨algorithm which enters request
  at end of queue⟩
else by ⟨send warning 'request ignored'⟩
else error
```

Fig. 2 Recovery block example

problems for the program requesting the transfer, at least the rest of the system is allowed to proceed without disruption. If this alternate fails, indicating that the queue was inconsistent when the recovery block was entered, then recovery has to take place at a more global level.

It should be noted that while the recovery block scheme enables redundancy to be specified at the algorithmic level in programs, it does not provide for redundancy in the data structures of programs. Thus, while an alternate can define any data structures local to its environment, the structures which are global to the recovery block must be fixed and their structure invariant. Therefore, there may be situations in which the static structure of global data adds to the problems of designing alternates.

Q4. It is common in fault tolerant hardware systems that a component is replaced when it fails. Does the recovery block scheme provide a software equivalent to this?

A4. An analogy can certainly be drawn between the replacement of faulty hardware components and the replacement of faulty alternates in the recovery block scheme. Borgerson (1973) has defined two terms for fault tolerant hardware: spontaneous replacement, in which the failing component is detected and replaced by an identical component; and spontaneous reconfiguration, which results in some degradation of the system. The two different ways in which recovery blocks can be used, as discussed above, could be described as providing spontaneous replacement and spontaneous reconfiguration at the software level. However, two points should be noted: firstly, a hardware component is usually replaced with one of identical design and construction—this is not usually the case with alternates. Secondly, the replacement of a hardware component is usually permanent; the replaced component may be repaired, but then kept as a standby-spare until needed. With recovery blocks, however, the failing alternate is only temporarily replaced, just for that execution of the block. On subsequent entries to the block that alternate will again be used in the hope that the new set of inputs does not cause the fault to manifest itself again.

The use of alternates of differing design and construction can be contrasted with another common feature in fault tolerant hardware systems, namely triple modular redundancy (and its variants). In TMR systems three identical components and voting circuits which examine the outputs from the components are used in order to mask the effects of any single component failure. In theory, a TMR system could be used to provide a means of tolerating design faults, as discussed by Avizienis (1975). This would involve the provision of three different versions of each component which, although designed independently, would all be intended to produce identical answers, preferably all at the same speed. The utility of such a scheme seems limited.

Q5. How should acceptance tests be designed?

A5. The acceptance test is a programmer-provided error detection mechanism which provides a check on the results of an alternate at the last possible moment, that is just before the recovery block is left and a set of recovery data is discarded. Clearly, a programmer can provide as little or as much checking as he considers necessary. Ideally, the acceptance test should test for the absolute correctness of the results. However, even if such a strict test could be designed, it may not be appropriate for four reasons: (*a*) because of performance considerations; (*b*) because the test for correctness may involve objects external to the computing system—for example, a stock control data base system may not be able to check its internal representation of the stock level against that actually in the warehouse; (*c*) because the alternates provide an increasingly degraded service and hence their results will not be exactly the same; and (*d*) because the likely complexity of such a test would make the acceptance test prone to design faults which would detract from the usefulness of the recovery block by rejecting correct results, or causing their rejection through the occurrence of errors during the execution of the acceptance test.

```
ensure ⟨true⟩
by      ensure ⟨best acceptance test⟩
        by ⟨best algorithm⟩ else error;
else by ensure ⟨next best acceptance test⟩
        by ⟨next best algorithm⟩ else error;
 . . .
 . . .
else error;
```

Fig. 3 Multiple acceptance tests

Thus in general the acceptance test will, as its name suggests, be a test on the acceptability of the results of the alternate rather than a test of their absolute correctness. For example, an acceptance test on a sorting algorithm might only check that the sorted elements were in order and their checksum was equal to the original value, but because of performance considerations would not check that any items from the original set had been modified or lost.

When the alternates of a recovery block have been designed to provide gracefully degradable software it is clear that the acceptance test can only be as rigorous as a check on the results from the weakest alternate. This has led some people to suggest that there should be a separate acceptance test for each alternate. Such a structure can be easily obtained by nesting recovery blocks, as illustrated in **Fig. 3.**

While this structure may appear satisfactory in isolation, it must be recognised that, in general, a recovery block will form only part of a program and that the acceptance test provides a check on the consistency of the results which are to be used by the rest of that program. (Indeed, it can be argued that the acceptance test should be the first part of the recovery block program to be designed.) Hence, it is likely that a single test of acceptance will often be required whether the alternates produce the same or different results.

The recovery cache mechanism can provide some run time assistance which may aid the design and implementation of acceptance tests: firstly, it can enable the prior values of objects to be referenced, so that the acceptance test can compare the current state with that on entry to the recovery block; and secondly, it can be designed to monitor the behaviour of the acceptance test with respect to the variables that had and had not been updated by an alternate. For example, it could raise an error condition if the acceptance test did not access all of the variables that had been updated by an alternate—this can ensure that the acceptance test performs at least some minimal checking of the new states of all updated objects and enables unintended updates to be detected.

The design of acceptance tests is a difficult area and still requires further research. While acceptance tests for specific problems can usually be specified, it is not yet clear whether a general methodology can be obtained, although there is some hope that the proof-directed methodology suggested by Anderson (1975) will provide some guidelines. It may also be noted that while the acceptance test is important, it will not be the only error detection mechanism in the system. As discussed previously, the underlying machine will provide mechanisms to detect errors in the execution of the program containing recovery blocks. Further programmer-provided checks could be incorporated into the alternates by means of *assert* statements, which raise an error condition if an error is detected. (Indeed, the structure depicted in Fig. 3 can be obtained through the use of *assert* statements instead of the nested recovery block, as described by Shrivastava and Akinpelu (1977).)

Q6. Can the recovery cache provide backward error recovery for all of the objects provided by the underlying machine?

A6. It is likely that there will be objects on the interface presented by the underlying machine for which backward error recovery is not available (for instance, the pages on a disc) or appropriate (for instance, objects shared by parallel processes). One method of dealing with such unrecoverable objects is to construct *multi-level* systems, whereby a new interface is constructed by software to provide new recoverable objects which are abstractions of unrecoverable objects. The implementation of the recovery for these new objects, although achieved by programmer-provided actions, will be transparent to the programs running on the new interface and extensions to the recovery cache mechanism can ensure that this recovery is automatically invoked as required. Two systems have been constructed at Newcastle demonstrating this approach. In the system described by Verhofstad (1977), the unrecoverable disc pages provided by a machine are used to provide a recoverable filing system for user programs. The second system (Shrivastava and Banatre, 1978) provides backward error recovery for processes sharing data for the purpose of competing for the resources of the system. It is beyond the scope of this paper to describe the implementation of such multi-level systems. The interested reader is referred to the paper by Anderson, Lee and Shrivastava (1977) which describes a conceptual model of recovery in such multi-level systems.

Q7. What happens if the recovery cache fails?

A7. In any fault tolerant system there have to be some components which are reliable in that the correct operation of these components is necessary for the correct operation of the fault tolerant aspects of the system. For hardware systems, such components are referred to as the 'hardcore'. The recovery cache is a major part of the 'hardcore' for the recovery block scheme, and it is assumed that its operation will be reliable. There are two justifications for placing so much reliance on the recovery cache: firstly, it would appear that the design of the recovery cache is sufficiently simple that standard hardware design practices can ensure that there are no residual faults in its design. The second justification is that in such circumstances any hardware component can be made as reliable as is necessary, through the application of fault tolerance techniques —cost is usually the only limiting factor. The recovery cache should only be a small part of a complete system, and hence the cost incurred in making it reliable should be acceptable.

Q8. What are the run time overheads involved in the use of recovery blocks?

A8. As with any system that provides redundancy and fault tolerance, the use of recovery blocks incurs run time space and time overheads which may not be present in fault intolerant programs. (It must be noted that the costs involved in a priori testing and validation of reliable fault intolerant programs may be substantial, and have led Hecht (1976) to suggest the adoption of the recovery block scheme to reduce these costs.) The space overheads for programs using recovery blocks stems from the extra storage required for the alternates, the acceptance tests and for use by the recovery cache. As discussed previously, the recovery cache can record a minimum of recovery data, which it is hoped will in general be a small percentage of the data space of a program. Shrivastava and Akinpelu (1977) report on experiments in which the figures for programs containing a single recovery block were between 3% and 39% (with an average of 17%), which are considerably less than the 100% overhead that recording the complete data space of the program would have entailed.

The execution time overheads required to support recovery blocks will depend on the time required to evaluate the acceptance test and on the recovery cache implementation. As discussed previously, the acceptance test overhead is the responsibility of the programmer, although Kim and Ramamoorthy (1976) have proposed an architecture which attempts to mitigate this overhead. The overheads imposed by the recovery cache will depend in the main on the implementation of the mechanism used to record recovery data. There are many implementations known for this function, each of which has different tradeoffs. The algorithm discussed previously, which records the old values of objects just before they were updated, optimises the normal progress of a program at the expense of the extra time required to restore the state if recovery is invoked. An algorithm which inhibited the update of an object, and recorded the new value of the object in the recovery cache would optimise the time required for recovery. (Indeed, with this organisation the recovery cache could act as a high speed buffer store and also possibly increase the speed of the normal execution of the program.) Also, at the expense of extra space, the time to record recovery data can be minimised or vice versa, as exemplified by the schemes described by Horning *et al.* (1974) and Anderson and Kerr (1976). It must also be noted that the recovery cache is intended to be provided as part of the underlying machine (for example, to be built in hardware) and should therefore be fast, particularly as some of its operations could be performed in parallel with the execution of the program. Thus it is felt that for a given set of constraints, a suitable implementation of the recovery cache can be specified. The programmer also has some control over the recovery time—recovery blocks can be nested to provide as fine a grain of recovery as is desired so as to minimise recovery time, at the expense of course of increased recording of recovery data.

In all of the above mechanisms the execution speed of the majority of instructions provided by the underlying machine will not be affected at all by the recovery block scheme. Apart from the instructions specific to the utilisation of recovery blocks (for example, start recovery block, end recovery block) the only instructions incurring any extra overhead will be those that write to the objects of a program and therefore require intervention by the recovery cache mechanism. Indeed, further optimisations can be applied so that only those instructions which write to an object that is external to an alternate are intercepted.

Although the overheads of a given mechanism can be quantified, it is difficult (impossible) to quantify the increased reliability that is obtained through the use of recovery blocks, since this is totally dependant on their effective deployment by the programmer. However, it is felt that the overheads associated with their use and implementation can be organised to be tolerable and acceptable.

Q9. Is the recovery block scheme the best technique for providing fault tolerant software?

A9. Exception handling (for example, as proposed by Goodenough (1975)) is often advocated as an alternative to the recovery block scheme. Exception handling can be thought of as a method of programming (forward) error recovery for *anticipated* faults. Thus for specific faults which can be anticipated and whose full consequences can be foreseen, exception handling can provide efficient recovery for it only involves correcting the known (anticipated) errors; in contrast, backward error recovery involves complete state restoration (albeit efficiently implemented by the recovery cache), not just restoration of the erroneous parts. However, the recovery block scheme can provide tolerance against unanticipated faults, while backward error recovery need make no assumptions about the fault and the damage it may have caused, and is in consequence a general recovery technique. Thus recovery blocks and exception handling techniques should be regarded as complementary rather than competitive approaches to achieving fault tolerant software. These topics are discussed further by Melliar-Smith and Randell (1977) who also present an example of a program combining both methods, using exception handlers to deal with simple anticipated faults, such as invalid input data, while utilising recovery blocks to deal with unanticipated faults, including those in the exception handlers themselves.

Conclusion

This paper has discussed the concepts and implementation of the recovery block scheme and has attempted to answer the questions which most frequently arise in discussions of the scheme. There is no other scheme known to the author which attacks effectively the area of fault tolerant computing that recovery blocks address, namely the tolerance of unanticipated design faults, particularly in the software level of a system. Experimentation with the implementation and utilisation of recovery blocks is being continued both at Newcastle and elsewhere and should shed further light on the scheme and determine its actual effectiveness.

Acknowledgements

This paper owes much to the work of past and present members of the Science Research Council sponsored 'Reliability Project' at the Computing Laboratory, University of Newcastle upon Tyne. In particular, I would like to thank Tom Anderson and Brian Randell for critical readings of earlier drafts of this paper.

References

ANDERSON, T. (1975). Provably Safe Programs, Technical Report No. 70, Computing Laboratory, University of Newcastle upon Tyne.

ANDERSON, T. and KERR, R. (1976). Recovery Blocks in Action: A System Supporting High Reliability, *Proceedings of Second International Conference on Software Engineering*, pp. 447-457.

ANDERSON, T., LEE, P. A. and SHRIVASTAVA, S. K. (1977). A Model of Recoverability in Multi-level Systems, Technical Report No. 115 Computing Laboratory, University of Newcastle upon Tyne, to appear in *IEEE Transactions on Software Engineering*.

AVIZIENIS, A. (1975). Fault-Tolerance and Fault-Intolerance: Complementary Approaches to Reliable Computing, *Proceedings of 1975 Conference on Reliable Software*, pp. 458-463.

BORGERSON, B. R. (1973). Spontaneous Reconfiguration in a Fail-Softly Computer Utility, *Datafair 73*, pp. 326-331.

FISCHLER, M. A., FIRNSCHEIN, O. and DREW, D. L. (1975). Distinct Software: An Approach to Reliable Computing, *Proceedings of Second USA-Japan Computer Conference*, pp. 573-579.

GILB, T. (1974). Parallel Programming, *Datamation*, October 1974, pp. 160-161.

GOODENOUGH, J. B. (1975). Exception Handling: Issues and a Proposed Notation, *CACM*, 18, 12, pp. 683-696.

HECHT, H. (1976). Fault Tolerant Software for a Fault Tolerant Computer, *Software Systems Engineering*, Online, Uxbridge, pp. 235-248.

HORNING, J. J., LAUER, H. C., MELLIAR-SMITH, P. M. and RANDELL, B. (1974). A Program Structure for Error Detection and Recovery, *Lecture Notes in Computer Science 16*, Springer Verlag, pp. 177-193.

KIM, K. H. and RAMAMOORTHY, C. V. (1976). Failure-Tolerant Parallel Programming and its Supporting System Architecture, *AFIPS Proceedings* Vol. 45, pp. 413-423.

MELLIAR-SMITH, P. M. and RANDELL, B. (1977). Software Reliability: The Role of Programmed Exception Handling, *Proceedings of ACM Conference on Language Design for Reliable Software*, pp. 95-100.

RANDELL, B. (1975). System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, Vol. 1, pp. 220-232.

RANDELL, B., LEE, P. A. and TRELEAVEN, P. C. (1978). Reliable Computing Systems, *Lecture Notes in Computer Science 60*, Springer Verlag, pp. 282-393.

SHRIVASTAVA, S. K. and AKINPELU, A. A. (1977). *Fault Tolerant Sequential Programming*, *Digest of Papers*, FTCS 8, pp. 207-208.

SHRIVASTAVA, S. K. and BANATRE, J-P. (1978). Reliable Resource Allocation Between Unreliable Processes, *IEEE Transactions on Software Engineering*, Vol. 4, pp. 230-241.

VERHOFSTAD, J. S. M. (1977). Recovery and Crash Resistance in a Filing System, *Proceedings of SIGMOD Conference*, Toronto.