Abstract:

A brief account is presented of the recovery block scheme, together with a description of a new implementation of the underlying cache mechanism. The salient features of a proposed computer architecture are described, which incorporates this implementation and also provides a high level of detection for errors such as the corruption of code and data. A prototype system has been constructed to test the viability of these techniques by executing programs containing recovery blocks on an emulator for the proposed architecture. Experiences in running this system are recounted with respect to errors introduced by deliberate attempts to corrupt the system.


RECOVERY BLOCKS IN ACTION:
a system supporting high reliability

by

T. Anderson
and
R. Kerr

TECHNICAL REPORT SERIES

Series Editor: Dr. B. Shaw

Number 93

© 1976 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU.
ANDERSON, Thomas, 


Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory; 1976.

University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 93.

KERR, Ronald

Suggested classmarks (primary classmark underlined)

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

Suggested keywords

ERROR DETECTION
ERROR RECOVERY
RECOVERY BLOCK
RECOVERY CACHE
RELIABILITY
SOFTWARE FAULT-TOLERANCE

Abstract

A brief account is presented of the recovery block scheme, together with a description of a new implementation of the underlying cache mechanism. The salient features of a proposed computer architecture are described, which incorporates this implementation and also provides a high level of detection for errors such as the corruption of code and data. A prototype system has been constructed to test the viability of these techniques by executing programs containing recovery blocks on an emulator for the proposed architecture. Experience in running this system are recounted with respect to errors introduced by deliberate attempts to corrupt the system.

About the Author

Dr. Anderson is a Senior Research Associate at the Computing Laboratory of the University of Newcastle upon Tyne.
RECOVERY BLOCKS IN ACTION:
A system supporting high reliability

T. Anderson and R. Kerr
Computing Laboratory
University of Newcastle upon Tyne, U.K.

Keywords and Phrases
Error detection, error recovery, recovery block, recovery cache, reliability, software fault-tolerance.

Abstract
The need for reliable complex systems motivates the development of techniques by which acceptable service can be maintained, even in the presence of residual errors. Recovery blocks allow a software designer to include tests on the acceptability of the various phases of a system’s operation, and to specify alternative actions should the acceptance tests fail. This approach relies on certain architectural features, ideally implemented in hardware, by which control and data structures can be retrieved after errors.

A brief account is presented of the recovery block scheme, together with a description of a new implementation of the underlying cache mechanism. The salient features of a proposed computer architecture are described, which incorporates this implementation and also provides a high level of detection for errors such as the corruption of code and data. A prototype system has been constructed to test the viability of these techniques by executing programs containing recovery blocks on an emulator for the proposed architecture. Experiences in running this system are recounted with respect to the execution of programs based on erroneous algorithms and also with respect to errors introduced by deliberate attempts to corrupt the system.

Introduction
Complex computing systems can never be guaranteed to be entirely error-free. Techniques by which systems can be made to withstand the effects of errors and continue to provide acceptable service are therefore necessary. There are three key issues involved in providing an acceptable service in the presence of errors. These are:
- the ability to detect errors before an intolerable degree of damage is incurred,
- the ability to discard faulty information arising from the error and to retrieve a valid system state,
- the ability to continue, with the expectation that further useful work can be performed.

A group of research workers at Newcastle is actively engaged in developing techniques which bear upon these issues from two complementary standpoints, namely architectural and methodological. A system embodying the architectural features in order to support programs which adopt a methodological approach to the attainment of reliability has been implemented.

This paper describes techniques developed by the Newcastle reliability group and recounts our experiences in running the system.

Recovery Blocks
Recovery blocks have been proposed (Hornung et al. (1974)) as a notation by which a programmer can make provision in the design of his program for checks on the acceptability of intermediate stages in the execution of the program, and also for alternative courses of action should these checks prove negative. Ways in which recovery blocks can be used in systems which aim to provide software fault tolerance have been reported (Randall (1975)), as has a proof-guided methodology for constructing the checks for acceptable program behaviour (Anderson (1975)).

Because recovery blocks play such a central role in our approach to obtaining greater program reliability, this section provides a summary of the basic recovery block scheme.

A recovery block may be represented as:

\[
\text{ensure} <\text{acceptance test}>
\]

by \(<\text{1st (primary) alternate}>

\text{else by} <\text{2nd alternate}>\]

\text{else by} <\text{nth alternate}>\]

\text{else error}

where \(n \geq 1\). The acceptance test yields a logical value, and its evaluation should have no side effect. All of the alternates are statement lists. Each alternate is executed in turn until the acceptance test holds. Before an alternate is entered the state of the program is set to what it was on entry to the recovery block.

A more precise description can be given by considering the recovery block as a means of providing recovery from detected error conditions. An error condition is raised whenever the underlying system detects an erroneous situation, such as an attempt to divide by zero or to reference an invalid memory location. Moreover, if an acceptance test is evaluated and yields the value false then an erroneous situation has arisen and the system will again raise an error condition. Whenever an error condition is raised it is recorded by the system to form an
experimentation. The investment of effort involved is considered inappropriate for a system still under development. Even the performance of a carefully designed set of trials will probably be postponed until a more complete system has been constructed, incorporating modifications suggested from consideration of the present system to extensions deriving from current research into, for example, recoverable type structures. What can be done is to observe, at least qualitatively, the extent to which the prototype system meets its basic design aim of providing a high level of recoverability. Our experiments to date have consisted of running on the emulator programs which were believed to be correct, programs which were known to contain source errors and programs whose object code had been corrupted deliberately.

This section merely attempts to summarise our initial experiences in using the prototype system, and to report any conclusions that can be drawn. It is stressed that these experiences are mainly drawn from the (fairly haphazard) testing and experimentation the system has undergone during its development, and are in no sense part of any controlled experiments on the system.

When an EML program is interpreted by the emulator, one possible outcome is that the program executes satisfactorily without any erroneous situations being detected, and hence without making any use of recovery facilities. That this did happen on a number of occasions is perhaps more a consequence of the elementary nature of various test programs than of any programming skill possessed by the authors.

A second possibility is that the program executes unsatisfactorily in that the results generated by the program are unacceptable to the writer of the program. One explanation for such an outcome is that the program is itself defective in one of the following ways:

in certain recovery blocks, no alternate compares acceptable results,

acceptance tests are too weak enabling results generated by faulty alternatives to be accepted.

If the fault does not lie in the program then the presence of a bug in the emulator has been detected. For instance, if the system collapses with a forced return to its basic support system, the unsatisfactory behaviour can be blamed on the emulator.

Not surprisingly, in view of the circumstances, many of the test programs initially executed unsatisfactorily. Any extension of the emulator and SUE.EML compiler carried the risk of introducing new errors into the system, some of which could be revealed in subsequent testing. Only rarely could unsatisfactory behaviour be attributed to the program, and on those occasions when it could the fault clearly lay with the programmer rather than with the recovery block approach.

Of much more interest is the third possibility, namely that the program executes satisfactorily despite the detection of one or more erroneous situations. Detection can either be due to the extensive basic error checking provided by the system, or to an acceptance test included by the programmer not being satisfied. A more useful categorisation is obtained by considering the error to have one of three possible sources.

(1) A planned inadequacy in the program.
(2) An unanticipated inadequacy in the program.
(3) An inadequacy of the system.

These are examined in turn, with examples drawn from the two programs outlined in figures 13 to 15.

do n:=0 to 99;

ensure i^2<=n and (n+i)^2>n and n=prior n

by try previous value of i

else by calculate a value for i using Newton's method

else by step i upwards from zero until (ii)^2>n

else error;

write n;i

end Figure 13: The program SORT

n:=read;
do i:=1 to n;

A(i):=read;

end

ensure A(i+1)>=A(i) for j=1,...,n-1

and \( \sum_{j=1}^{n} (A(j) - prior A(j)) = 0 \)

by order the values in A using Shell's sorting method

else by order the values in A using linear selection

else error;
do i:=1 to n;

write A(i)

end Figure 14: The program SORT

ensure A(i):=prior A(j) and A(j):=prior A(i)

by A(i):=A(i) - A(j); A(j):=A(i) + A(j);

A(i):=A(i) - A(j)

else by w:=A(i); A(i):=A(j); A(j):=w

else error

Figure 15: The recovery block EXCHANGE

(1) Planned program inadequacy

In an attempt to test the system which occurs when a program contains a genuine bug, programs containing deliberate mistakes were executed on the emulator.

Figure 13 shows the program SORT which has been used extensively to test the emulator. This program is intended to print out a table of integer approximations to the square roots of the numbers 0 to 99. Each alternate of the recovery block attempts to place the correct value in the variable 1. The acceptance test checks explicitly that it contains the largest integer not greater than \( \sqrt{n} \), and that the value of n has not been changed.
The primary alternative in SORT is based on the principle that the largest integer not greater than \(\sqrt{n}\) might well be equal to the largest integer not greater than \(\sqrt{n-1}\), and so does nothing at all. Unfortunately, whenever the value of \(n\) is a perfect square this does not succeed, and in these cases the acceptance test rejects the erroneous value left in \(i\). Furthermore, when \(n\) is zero at the first execution of the recovery block, the variable \(i\) has never been used and has no previous value. In this case an error condition is raised during the evaluation of the acceptance test since the emulator rejects any attempts to use the value of an uninitialised variable. This rather frivolous primary alternative was added to an earlier version of the SORT program in order to exercise the recovery mechanisms of the emulator. Although clearly defective in general, on those many occasions when it is successful the primary alternative is exceedingly efficient. There may be some justification, in particular contexts, for employing a primary alternative for which it is known that there exist circumstances in which it will fail, if for most cases the alternative has the virtues of simplicity and efficiency. Reliability may still be enhanced as a result of the alternative's simplicity and independence from the other alternatives.

The second alternative in SORT employs a conventional root finding algorithm, modified to use integer arithmetic. It was known that the encoding of the algorithm is such that when \(n\) equals zero the value computed is incorrect, and should therefore be rejected by the acceptance test. Unlike the faulty primary alternative, this error did in fact arise from a genuine programming mistake and so may be regarded as a little more authentic.

The third alternative is programmed very simply with the aim of increasing dependability at the expense of efficiency and is expected always to be able to satisfy the acceptance test.

Next, consider the second example program, SORT, given in figure 14. This program is intended to read in a value for \(n\), then read \(n\) integers, sort them into ascending order in the array \(A\), and then print them out in order. The alternates of the recovery block implement different well-known sorting algorithms, and both make use of the recovery block EXCHANGE shown in figure 15. The acceptance test confirms that the entries in the array are in ascending order and attempts to verify, by means of a sum check, that no values have been changed.

The primary alternative in SORT is an encoding of Shell's sorting method. The encoding was obtained by very cautiously translating an ALGOL 60 version (taken from a programming manual) into BBC-EML in the hope and expectation of introducing an error. In fact two errors were made. The first of these had the effect of assigning the value zero to a variable declared to be always positive. Detection of this by the emulator led to the second alternative being invoked. After correcting this error, a second bug was uncovered (the step component of a FOR statement had been omitted) which has the effect of leaving the array only partially sorted. Unless the integers are almost in order initially they are left out of order - which is detected by the acceptance test.

The second alternative is a straightforward encoding of linear selection and is thought to contain no errors.

The SORT program contains a nested recovery block EXCHANGE, used by both alternatives to interchange the values of \(A(i)\) and \(A(j)\). The acceptance test checks explicitly that the values have been interchanged. The second alternative is just the usual cyclic exchange using an auxiliary work variable, but the primary alternative exchanges the values without using an additional variable. In addition to its drawbacks of slowness and obscurity, the primary alternative is interesting in that it may fail because of overflow (very easily in fact since the array elements may be declared to hold values from a fixed integer subrange) and this is why it was chosen.

Both of these programs, SORT and SORT, are executed correctly by the emulator; despite their deficiencies they produce the desired results.

Another approach to modelling the effects of genuine programmer errors is to corrupt the code segment of an otherwise satisfactory program and the emulator is equipped accordingly with a special mechanism for injecting errors. Given that the program contains provision for error recovery, there is some prospect that the program will continue to give service. With more stringent provisions much more can be claimed. Consider again the SORT program, which has been subjected to considerable corruption in this way. The recovery block in SORT has the property that its final alternate is designed in such a way that the program has a very high degree of confidence that it will always pass the acceptance test, and also has the property that the acceptance test is complete in the sense that passing the test in itself guarantees satisfactory results (this last is not usually attainable except at prohibitive cost, e.g. the sum check in SORT). So, for the SORT program, the stringency of the test and confidence in the final alternate lead to the claim that no matter what is encoded in the earlier alternatives the program will still run satisfactorily. As a corollary to this claim, arbitrary corruption of the code fragments corresponding to the primary and second alternates of SORT should not prevent the program from completing its execution successfully.

Project members and visitors to the department have risen to the implicit challenge and with ingenious penetration techniques attempted to refute this claim. Although it has to be conceded that their efforts have on rare occasions met with some limited successes, a considerable time has elapsed since the program last failed to produce the now rather tedious table of square roots. About 100 bytes of EML code are available for modification. Random changes are almost always detected immediately by virtue of the redundancy retained in the EML code. More sophisticated and structured attacks, probing for weaknesses in the emulator's implementation, are usually employed. Even so, in the vast majority of cases, the recovery mechanisms have successfully detected and recovered from any damage done to the system, often to the annoyance (initially) of the authors. The damage was often much more extensive than was anticipated, largely because of the difficulty in anticipating beforehand all of the implications which a change made at the EML level could have, particularly when the change impinged directly on the recovery mechanisms. When a successful penetration has been made this has been, with one exception, attributable to an implementation error which was then rectified. Errors detected in this way have usually been rather obscure. The exception involved exploitation of a weak acceptance test in an earlier version of SORT written before prior was available.

The technique of modifying the EML code to introduce errors into a program has been considered as a form of planned program inadequacy, which it certainly is. However, as has been mentioned, the errors
so introduced are rarely as well understood as those built into the source version of a program. Consequently the detection and recovery mechanisms are tested against situations which are not fully anticipated; since the erroneous situations which arise as a result of authentic programming errors cannot be studied in advance, the success of the recovery techniques during these experiments is encouraging.

(ii) Unanticipated program inadequacy

A principle aim of the techniques provided by the prototype system is to help programmers construct defences against the residual programming errors, known to remain in complex software systems. Some doubt must attach to conclusions drawn from observing the success of the recovery techniques in handling more or less contrived errors in small test programs, but this is unavoidable until experimentation on a larger scale can be contemplated.

It is therefore pleasant to record one example of a completely un planned programming error which was handled successfully by the system. It was not realised that the primary alternate of the EXCHANGE recovery block is completely defective for the special case i, when the two variables to be exchanged are in fact the same variable. In this situation the effect of the primary alternate is to set the variable A(i) to zero. The acceptance test detects this to be erroneous (unless prior A(i) = 0), the value of A(i) is restored and the second alternate correctly exchanges A(i) with itself.

Whether this authentic example of a programming error is in any real sense a better test of the recovery techniques than the more contrived examples presented earlier is highly dubious, but it was certainly rewarding to observe (after the event) that the prototype system had functioned as intended and recovered from a totally unexpected error. The error itself is typical of manyprogramming mistakes—a special case or unusual combination of circumstances is so often overlooked.

(iii) System inadequacy

Because the emulator program is itself a substantial piece of software, the presence of errors in the system is not surprising, particularly in view of the numerous extensions and modifications which have been made in the course of its construction. It has already been stated that errors in the emulator often led to unsatisfactory behaviour of a program being interpreted. Of much greater interest were a number of occasions on which an error in the emulator resulted in considerable corruption of the state of a program being interpreted but which was then detected as being erroneous and successfully recovered from. In fact, if the faulty emulator continued to observe the constraints imposed by its own basic support system (otherwise the emulator collapsed) then a satisfactory program behaviour was usually achieved. Data corruption due to faulty emulator behaviour must be confined to data defining the state of the interpreted program, and that program must contain adequate recovery provision if it is to execute satisfactorily. In these circumstances the emulator is able to recover from its own deficiencies because of its ability to permit the program being interpreted to recover.

Two typical examples are recounted which occurred during penetration attempts on the program SQR.

It was discovered that the emulator did not ensure that expression evaluation was confined to temporary locations at the head of the data stack. By modifying the ESL code of SQR so as to set up a series of "add" instructions (which overwritten the data stack pointer) it was possible to delete from the data stack variables declared in the current block. When an attempt to access a deleted variable raised an error condition, the variables were restored from the recovery cache which, of course, records all relevant changes made to the data stack, irrespective of the reason for the change.

The second example was a consequence of the emulator not checking for overflow of the control stack. A control transfer was altered to transfer control to a code fragment containing the transfer, thus creating a recursive loop (with some difficulty because of the need to match transfer and fragment). Each pass around the loop placed additional entries on the control stack, which eventually overflowed, overwriting the data stack display. When the display was next used an error was raised, followed by the restoration of data stack and display, and by the retraction of the control stack.

In summary, we have been very gratified by the extent to which the prototype system has met its basic design aims. Error recovery has been both automatic and accidental programming errors, and from faulty behaviour due to errors in the emulator (on one occasion the fault in the emulator was due to an error in the SUII compiler) has been impressive. The most important feature of the facilities supported by the system seems to be that they can be used to implement recovery capability which is not designed to meet specific error situations, and as a result is able to deal with the unexpected situations caused by errors.

The detailed design decisions entailed in implementing a system which provides error recovery have led to a better understanding of the basic needs for error detection and recovery facilities. Two points can be made here. It now seems preferable to regard the failure of an acceptance test as a run-time error rather than to consider a run-time error as equivalent to the premature failure of an acceptance test. The former view leads to a cleaner and more secure implementation, and has been adopted throughout this paper and in the prototype system. Secondly, we now see that a complete separation could be made between the recovery structures and the other structures (data and control) defining the state of a program. If this were done, recovery could be made more comprehensive and more uniform than is the case with the present system.

A limitation of the basic recovery scheme implemented by the prototype system is that it provides recoverability for a single sequential process in isolation from any other processes with which it interacts. Some progress has been made in removing this restriction (Randell (1974)) and research is continuing with the aim of extending the recovery techniques to a set of mutually dependent processes. However, the present implementation is not likely to be extended in this direction. Rather we would wish to place on the experience gained from building and using the prototype system in any future, more general, implementation.

Acknowledgments

The work described here was carried out at the University of Newcastle upon Tyne as part of a research project entitled "The Design of Highly Reliable Computing Systems". This project is funded by the
Science Research Council of Great Britain whose interest and support are greatly appreciated. We acknowledge also the enthusiastic and good-humoured co-operation of all our colleagues on this project.

References


