A Program Structure For Error Detection And Recovery

J.J. Horning *(University of Toronto, Canada)*, H.C. Lauer, P.M. Melliar-Smith, B. Randell *(University of Newcastle upon Tyne, England)*

ABSTRACT

The paper describes a method of structuring programs which aids the design and validation of facilities for the detection of and recovery from software errors. Associated with the method is a mechanism for the automatic preservation of restart information at a level of overhead which is believed to be tolerable.

1. Introduction

Prior research into reliable computing has concentrated on the reliability of the hardware, on the detection of hardware errors and on the configuring systems to allow continuation of service in the presence of hardware errors. But observation of prevent-day large systems indicates that software faults represent a problem whose significance is at least as great as that of the hardware faults. Whilst conceding the importance of current research on improving the quality of software (e.g. work on program "correctness proofs"), the present paper is based on the view that it is also worth providing error detection and recovery facilities for both hardware and software errors. In what follows we will concentrate on software errors although we believe much of our work is of equal relevance to many types of hardware errors.

This paper describes the recovery block concept, a method of structuring programs which is aimed at aiding the design of error detection and recovery facilities, and the recursive cache, an associated mechanism which provides means for automatic "back-tracking" at a level of overhead which is believed to be tolerable.

2. Error Detection and Recovery

Reliable operation of a computing system depends on both error detection and error recovery. Several classes of error detection techniques are available, some of which operate automatically on an instruction-by-instruction basis while others take a broader view of correct operation based on programmed checks and assertions. It is characteristic of error detection that several techniques can readily be used together, and the recovery block concept aims to provide error recovery after any kind of detected error.

Recovery and restart of the program after error detection is a difficult problem. Where instruction-by-instruction error detection is provided, the number of possible errors is too great to provide explicit recovery action for each possible case, while any automatic recovery operation which simply aims to repair the state of the program so as to allow its continuation in a valid manner cannot be expected to achieve correctness rather than mere legality. Alternatively, on detection of an error by a programmed check, the number of possible ways in which the program may have erred is very large and obscure side-effects of the error may have spread into the system. The analysis, with certainty, of such an erroneous program state is in general beyond our capabilities, and therefore repair of the program state cannot be recommended.

For many applications, frequently the applications requiring the highest reliability, error recovery by repetition of whole job steps or other major program units must imply a substantial recovery overhead and a degradation of service. Such applications require a recovery mechanism which can achieve local recovery from an error whenever possible.

It is characteristic of error recovery operations that they are very much more error prone than the main programs, and are very difficult to check. It is an objective of the recovery block concept that the error recovery actions should be testable and that they should be checked with the same rigour as the main program. It is of course obvious that repetition of the operation with the same program will not always achieve recovery from a program error.

The recovery block concept is aimed at the provision of a well-structured context for explicitly considered error recovery operations. These can be constructed at various levels within the program so that if a local recovery operation should fail a more global recovery action can be substituted. An other aim is that the structure should make explicit the nature of the checks which are applied and also the nature of the action to be taken in the event of error, the number of such actions being strictly limited.

3. Recovery Blocks

A well-structured program is constructed from identifiable operations, many of which are themselves constructed from further smaller operations. The proposed scheme is based on the selection of a set of these operations upon which to base the recovery operations. These will be referred to as Recovery Blocks. Recovery blocks can be nested like Algol blocks, but a recovery block must have a more complex internal structure than an Algol block so as to support error recovery. Each recovery block contains a primary block, an acceptance test, and zero or more alternate blocks.

The primary block corresponds exactly to the block of the Algol-like program, and is entered to perform the desired operation.

The acceptance test is executed on exit from the primary block to confirm that the primary block has performed acceptably.

If the primary block is detected to be in error, the alternate block is entered and is required to perform the desired operation in a different way or to perform some alternative action acceptable to the program as a whole. The acceptance test is then repeated.

The diagram of a recovery block's structure given in Figure 1 is intended to be illustrative, rather than syntactically representative. The double vertical lines define the scopes of recovery block, while the single vertical lines define the scopes of primary and alternate blocks. Figure 2 shows how the primary and alternate blocks can contain, nested within themselves, further recovery blocks.

4. Acceptance Tests

The acceptance test is a section of program which is invoked on exit from a primary or alternate block. This acceptance test can be regarded as an assertion of the effects of the execution of the recovery block which are required for the correct operation of the surrounding program [1]. The acceptance test provides a binary decision as to whether the assertion is satisfied and thus as to whether the recovery block has been executed in a manner acceptable to the rest of the system. There is no requirement that the test be, in any formal sense, a check on the absolute correctness of the operation - it is for the designer to decide on the appropriate level of rigour of the test. For each recovery block there is a single acceptance test, invoked on exit from the primary and also on exit from the alternate should the alternate be required. Thus in figure 2, the acceptance test BT is invoked on completion of primary block BP, so as to check the acceptability of the results of BP.

The acceptance test does not lie within the primary block, but can be considered to be a part of the next enclosing block. Thus the acceptance test cannot access the local variables of the primary or alternate blocks: indeed there is no reason why the local declarations of these blocks should be the same. The function of the acceptance test is to ensure that the operation performed by the recovery block is to the satisfaction of the program which invoked it. The acceptance test is therefore performed by reference to the variables accessible to that program, rather than to local variables which have no effect or significance after exit.

The acceptance test can reference any variable within the current scope immediately enclosing the recovery block. The primary block may modify some of these variables. For convenience and increased rigour, the acceptance test is enabled to access such variables either for their modified value or for their original unmodified value.

5. Rejection by an Acceptance Test

There are four possible causes for the rejection of a primary or alternate block. These are:

a) an error within the block, detected explicitly by the acceptance test,

b) failure to terminate, detected by a timeout,

c) detection of an error within the block by one of the implicit error detection mechanisms (e.g. protection violation, divide by zero, etc.),

d) explicit or implicit rejection within an inner recovery block which exhausts the recovery capability at that level.

If the primary of a recovery block is rejected then the recursive cache mechanism invokes an alternate. When this alternate terminates, its results are submitted to the same acceptance test, and should the acceptance test satisfied, the program proceeds using the results generated by the alternate If the acceptance test is again not satisfied then a further alternate is. Thus, in figure 2, if the results of primary block BP are rejected by acceptance test BT, then alternate block BQ is invoked. If the results from BQ are unacceptable to BT, then BR must be invoked.

Should all the alternate blocks have been obeyed, and all have failed to satisfy the acceptance test, then the entire recovery block must be regarded as having failed. This causes rejection of the enclosing block which invoked the recovery block, and the alternate to that enclosing block must be attempted instead Thus, in figure 2, if the results from alternate block BR are still unacceptable to acceptance test BT, then recovery block B as a whole, and therefore primary block AP, must be regarded as having failed. The next program to be attempted is alternate block AQ.

If an error occurs during the execution of the program of an acceptance test, this is regarded as an error occurring within the next enclosing recovery block. The alternate invoked is therefore that for the enclosing block rather than that for the block whose acceptability is being tested. Thus if an error is detected whilst executing the program of acceptance test BT, the alternate block AQ must be entered.

6 Primary and Alternate Blocks

It is an objective of the recovery block concept that primary block can be written in exactly the same manner as in a conventional system. There are no restrictions on the operations which are performed by the primary blocks, no restrictions on the calling of procedures or the modification of global variables, no requirements for the explicit preservation of restart information, and no special programming conventions.

Similarly the intention is that the alternate blocks can be written in exactly the same manner. The design of an alternate block is not affected by the prior unsuccessful execution of the primary block. The recursive cache mechanism described below ensures that an alternate block is executed as though the primary had never been entered, and as though the alternate had been substituted for the primary in a conventional program structure.

When an alternate block is entered it must therefore be presented with exactly the same environment as was its primary when it was entered. All the operations of the primary must have been undone, all the non-local variables altered by the primary must have been restored to their previous values. It is a requirement on the mechanisation that this recovery be achievable with reasonable efficiency both for assignments as described below in section 8, and for more complex operations, as described in section 11. The recovery block scheme provides a structuring of programs in time and space so that the recovery information can be retained with greater facility and efficiency than would be possible with for instance, a core image checkpoint mechanism.

In particular it should be noted that the alternate has no access to the reason why the primary was rejected and no access to any results calculated by the primary. It is important that a record of each rejection is preserved for subsequent investigation, but it is envisaged that the records will lie beyond the scope of the program being run. There may be occasions when it would be convenient for the alternate to know what had gone wrong with the primary, but the number of possible error conditions is very large, and it is not always easy to distinguish one error from another. Since the alternate cannot be expected to categorise and accommodate each cause of error explicitly it would appear preferable that the alternate should always start again from the entry to the recovery block without any record of previous rejections. Errors which are expected to be sufficiently frequent that special handling would be appropriate can perhaps be regarded as normal program conditions rather than unforeseeable errors.

As mentioned earlier, an alternate block might perform the desired operation in a different way, presumably less efficiently, but perhaps by a simpler and less error-prone algorithm. However the more likely case is for an alternate block to be designed to provide an operation which, though less desirable, is still acceptable to the program as a whole. Thus the recovery block structure might be used as a means of structuring the rather ad hoc error recovery facilities that many existing systems incorporate [2].

7. 0rthogonality of Design

We believe that the recovery block concept should not impose any constraint on the programming or the architecture of the computer, for any design which imposes constraints may preclude, or be precluded by, other techniques intended to facilitate reliable computing.

Each primary and alternate block is programmed in exactly the same manner as a conventional program, written in any desired programming language and with any desired programming style or methodology. Extensions to existing languages are required only to define the recovery block structure, and to permit access to prior values of variables during acceptance tests.

It is natural to equate the recovery blocks with a subset of the blocks of an Algol-like program, but it is not necessary that all the Algol blocks be recovery blocks or even that the programming language provide an Algol-like block structure. The only requirements are that the recovery blocks should be explicitly defined, that they should be dynamically nested, and that entry to and exit from recovery blocks should be explicit. Clearly, well-structured programming techniques are to be encouraged, but the recursive cache concept does not depend on them. It is as applicable to programming in assembler on S/360 as to Algol on a B6700. Thus the recovery block concept is substantially orthogonal to programming languages and methodologies.

The recursive cache mechanism, described below, operates entirely with "words" in a single linear virtual address space. The user may program with various sizes of data, may use vectors, heaps, own variables, complex list structures, parameters, or even recursive procedures. It is assumed that the computer architecture will accommodate this variety and by a display, descriptors, indexing or other mechanisms will convert all store references into references to words in a linear virtual address space. The address of such a word will be known as a "virtual address". The mechanism operates entirely with these words and their virtual addressees and the significance of this data to the user program is of no concern to the cache mechanism, The description below is in terms of a classical Algol stack machine but it is believed that conventional computer architectures can readily be accommodated, and thus the error recovery scheme is orthogonal to the architecture of the computer.

8. The Recursive Cache Mechanism

The requirement that an alternate block be presented with exactly the same environment as was its primary could be mechanised by appropriate core image check points, (or even by retaining enough information to allow programs to be executed backwards, instruction by instruction [3]), but the associated overheads would defeat the purpose of the concept. Consequently a specialised mechanisation is presented which aims to reduce the overheads to an acceptable level, while remaining invisible to the user.

The heavy overhead of recording a conventional checkpoint on entry to each recovery block is caused by the recording of the entire context of the recovery block, a substantial quantity of information. But in many cases only a few of the non-local variables will be modified bythe recovery block, and all that would be needed to restore the environment are the virtual addresses and prior values of those variables which are actually modified [4]. Thus in this mechanisation only on writing into a "word" is the virtual address and previous value of that word recorded in an additional storage area to be called the cache.

Since only the values of the variables on entry to the recovery block need be preserved, it is appropriate to associate with each word a Boolean flag to indicate that the required value has already been preserved in the cache. All these flags are cleared on entry to the recovery block, and each writing operation tests the appropriate flag. If the flag is clear, then the virtual address and current value of the word must be recorded in the cache, and the flag must be set. If it is set, then no special action is required before the write operation.

Because the- recovery blocks are nested, the cache can be organised as a stack, as is shown in figure 3. Stack marks are placed in the cache stack to indicate recovery block entries and similar stack marks in the main stack facilitate the recognition of variables local to the current primary or alternate block The marks divide the main stack and cache stack into (possibly empty) regions. Each main stack word contains a value and a Boolean flag which (for non-local variables) indicates modification within the current recovery block. Each cache entry contains the virtual address of a word and the value of that word on entry to the recovery block. Note that the set of words flagged in the in stack corresponds exactly to the set of words recorded within in the top region of the cache stack.

The term recursive cache was originally introduced by (inaccurate) analogy to the cache of the IBM 360/85. In fact our usage of cache can be viewed as matching its dictionary meaning of a "hiding-place", since prior values of variables are hidden in the cache in case they might be needed again. In contrast IBM’s usage of this term, which they have now abandoned in favour of "buffer-store", was in relation to the high speed store of a demand paged virtual memory system; such a high speed store would contain current values of variables.

9. The Algorithm of the Mechanisation

On entry to a recovery block, stack marks are placed in the main and cache stacks, and all the flags are cleared. The flags will subsequently be reset by examination of the appropriate cache region and the same technique may be used to clear them. For examples see stages (b) or (c) in Figure 4.

Reading any variable (local or global, whether modified or not) is done by fetching the value of the word with the appropriate virtual address, stage (e) in Figure 4. The flag is ignored. Thus no overhead is incurred on reading, an important consideration.

Assignment to a local variable is also performed conventionally, important because many assignments are to local variables, stages (b) or (c) in Figure 4. Local variables do not need to be cached and thus the flag is of no significance.

Assignment to a non-local variable involves testing the flag. If on assignment the flag is found to be clear, then the flag must be set and an entry, comprising the virtual address of the word and its current value, is pushed onto the cache stack. The assignment is then performed, stages (d) or (e) in Figure 4. If the flag is set then the prior value has already been cached and the assignment proceeds conventionally, stage (f) in Figure 4.

The performance of an acceptance test implies exit from the primary or alternate block, and therefore the deletion of all local variables from the main stack. If the acceptance test involves access to prior values of variables, these values may be obtained by searching the top region of the cache stack, though it may be appropriate first to check the flag of the word in the main stack, where the required value will be if it is actually unchanged.

If the acceptance test should reject the block then the environment must be restored to exactly the situation existing on entry. This is done by popping each entry of the top region of the cache stack, and using its value to reset the word at the virtual address cited, the flag also being cleared, stage (h) of Figure 4. Note that this must clear all the flags, the condition established on entry to a recovery block. By removal of stack marks, this operation may be performed repeatedly to obtain recovery at an outer recovery block.

The action if the acceptance test is passed is more complex. The aim is to set up the cache and the stack as though the accepted block had consisted solely of its non-local assignments, some of which might of course be local to the enclosing block, others of which might have been preceded by assignments to the same variables during this enclosing block. First, all the flags are cleared. Then the entries in the top-but-one region of the cache stack are accessed and the flags are set for all the words those entries address. Next the entries of the top region of the cache are processed.

If the word addressed by the cache entry is local to the enclosing block then the cached value may be discarded as caching is not required for local variables. Similarly, if the flag of the word addressed is set the cached value may be discarded, for the value cached in the lower cache region is the true value of the word on entry to that recovery block.

If the flag is not set, then the cached value is the value that a non-local variable had on entry to the enclosing recovery block. The cache entry must therefore be included in the top-but-one region of the cache stack, and the flag must be set.

While the top region of the cache stack is being processed it may be necessary to enlarge the top-but-one region. It is therefore appropriate to process this top region in the reverse sequence (FIFO rather than LIFO). When the top region has been fully processed and the stack marks removed, processing may resume on the enclosing, now the current, recovery block. Stage (g) of Figure 4 shows a few of the possible circumstances. The main overheads of this implementation are incurred at block entry and block exit time, and will depend linearly on the number of different non-local assignments that have occurred. The scanning of the cache that is involved is comparatively simple, and is in our opinion quite appropriate for hardware implementation. The one instance of a search is that involved in accessing the prior value of a variable that has been changed, which occurs only in acceptance tests. The space needed in the cache will depend on the program structure, but for a given choice of recovery blocks no unnecessary information will be saved, a claim which would be very difficult to make for a system with programmer-specified check-pointing.

10. Assignments and Procedure Calls within Acceptance Tests

In the basic scheme, an acceptance test is a transparent operation whose sole effect is to provide a binary decision on the acceptability of a recovery block. In practice however an acceptance test may be a quite complex program containing assignments and procedure calls.

Because of notational problems, it would appear to be inappropriate to allow an acceptance test to contain a recovery block. But if the acceptance test is allowed to call procedures which contain recovery blocks, then the required effect is readily achieved. The use of procedures may be very desirable for the proper structuring of complex acceptance tests.

Only minor extensions to the basic recursive cache mechanism are required to accommodate procedure calls within acceptance tests. The most significant extension required is to allow an acceptance test to pass to a procedure, as a parameter, a reference to the previous value of a variable which has been altered within the recovery block. The parameter may be of course a complex data structure, only parts of which have been altered. It would appear necessary to extend the address or reference to the parameter with a recovery block level field, which can be carried over into any subsequently derived address or reference. This level field can be used to ensure recovery of the correct values from the cache.

Modification of these original values within an acceptance test is clearly reprehensible and without any possible justification. The appending of the level number to an address could therefore be used to render the data thus referenced read only.

11. Recoverable Procedures

The programs discussed above generated their results entirely by assignment to storage locations, and were recovered simply by reversal of those assignments. But many programming operations generate results other than by assignments, or need to preserve some results even when the processing of the operation is abandoned and an alternate is attempted. Typical examples are operations which involve file access and input-output interfaces, accounting routines, diagnostic traces, and interactive user interfaces. It is characteristic of such programs that error recovery is more complex than automatic reversal of assignments and an opportunity must be provided for the program designer to specify the appropriate recovery action.

It is proposed that operations requiring special recovery action should be structured into procedures, to be known as recoverable procedures. A recoverable procedure is not itself a recovery block with an acceptance test and alternates, though its body consists of a recovery block. Such a procedure may declare own variables, whose values are not automatically reset by the recursive cache mechanism in the event of error, but can be restored by program within the recoverable procedure. It may be appropriate to allow several recoverable procedures to be associated and to share a set of own variables. Such a structure would follow naturally from the classes of Simula [5] or the Type mechanism of Campbell and Habermann [6].

Just as the recovery of simple variables involves the saving of a value on first assignment within a recovery block, multiple further assignments without special action, and the restoration of the prior value in the event of error, so the recoverable procedure must provide three entry points, the save, the normal and the reverse entry points as is shown in Figure 3. The save entry point preserves such recovery information as the programmer specifies before performing the required action, and the mechanism will enter here the first time that the procedure is called within a recovery block Subsequent calls of the procedure within the recovery block will enter at the normal entry point. (The assumption is that the information saved on the occasion of the first call of the procedure suffices to allow the system to be reinstated to the satisfaction of the programmer, even if there are many further calls). It is important to note that the program which calls the recoverable procedure is not aware of the save entry point, which is invoked automatically by the cache mechanism. The reverse entry point is also invoked automatically by the cache mechanism, in the event of an error in the block that invoked the procedure, so as to provide an opportunity to restore the recoverable procedure in accordance with the information saved on entry through the save entry point.

An example- of a recoverable procedure might be a file access interface which maintains own variables indicating the current position of each file. The save operation would simply record the values of these variables on entry to each recovery block, while the reverse operation would use the saved values to reposition the files.

12. The Mechanisation of Recoverable Procedures

When the program enters the scope within which the recoverable procedure is declared, a descriptor providing access to the procedure is placed on the main stack, as are the own variables of the procedure. Both the descriptor and the own variables are provided with flags for use by the cache mechanism, just as any other variables on the main stack. An example is shown in Figure 5.

When the recoverable procedure is invoked for the first time, the flag on its descriptor will be found clear, indicating that the procedure has not been used previously within this recovery block. The cache mechanism now sets the flag and records the descriptor and its virtual address in the cache. It then enters the save entry point of the procedure. The recoverable procedure now obtains a further area of the cache stack within which to record its recovery information and enters its internal recovery blocks which perform the saving of recovery information and the function required of the procedure.

Subsequent calls on the recoverable procedure within the same recovery block will find the flag on the descriptor already set. The recoverable procedure will then be entered through the normal entry point with no special action. But should the program enter a further recovery block before again calling the recoverable procedure, then the flag on its descriptor will have been cleared and a further set of recovery information must be recorded in the cache stack to correspond to the new recovery block within which recovery may be required

On successful exit from a recovery block, the descriptor and its associated save area in the cache stack are treated by the cache mechanism just like any other variable recorded in the cache stack. They are transferred from one cache region to the next, resetting the main stack descriptor flag, until either they become local or the descriptor is already present in that cache region. Within the recovery blocks inside the recoverable procedure, the own variables of the procedure, and the variables within the save area in the cache stack, are regarded as non-local and can be restored just like any other variable in the event of error. But once the acceptance test of the first recovery block of the procedure is satisfied, during the exit from the recovery block these variables are regarded as local within the next recovery block, regardless of their position in the main stack, and their cached prior values are therefore discarded. Thus exit from the recoverable procedure effectively renders permanent any assignments to the own variables, and subsequent recovery can only be through the recovery program contained within the recoverable procedure. However any assignments made by the procedure to non-local variables, either directly or through some parameter mechanism, are treated normally and therefore can be undone by the cache mechanism.

After an error, the top cache stack region is processed in the conventional manner, and when the descriptor for the recoverable procedure is found, the cache mechanism invokes the reverse entry point of the procedure. The reverse entry point may involve a recovery block, further procedure calls and recovery blocks, and any other processing necessary to restore the own variables to appropriate values. (We have yet to consider the need for allowing assignments to other non-local variables from within a reverse operation.)

13 Exercising the Alternates

Within many systems containing error recovery mechanisms, the mechanisms are not tested with sufficient rigour. Adequate validation of a system's ability to cope with errors occurring in the midst of attempts to recovery from an earlier detected error can be particularly difficult. Thus errors remain within the recovery programs and the reliability of the system is adversely affected.

It is proposed that, in the recovery block system, provision should be made for the automatic exercising of the alternates, either all alternates being exercised or only a proportion on a probabilistic basis. Rejections would be recorded for subsequent analysis. This provision can be made either by an explicit mechanism, or else perhaps by appropriate programming of the acceptance tests.

14. Further research

The recovery block scheme is only in the early stages of development, and its value has yet to be demonstrated. At present, a programmed emulator incorporating a recursive cache mechanism is being implemented for our dual processor PDP11/45, and a variety of application programs, structured into recovery blocks, will be programmed to test the concept for both efficacy and cost. Subsequently we hope to extend the recovery block concept into the code of the operating system.

We have already received encouragement from some brief experiments carried out by a colleague, David Wyeth, which involved interpretive execution of a number of Algol W programs. Even regarding each block as being a recovery block, it was found that the amount of space that would be needed for a cache was in every case considerably smaller than that needed for the stack. It would of course be possible to design a program, and its recovery block structure, so that the cache size greatly exceeded that of the stack. We believe this to be unlikely in practice, but could imagine that a scheme for removing part of the cache to backing storage, which could take advantage of the simplified patterns of access to the cache, might be worthwhile.

The most significant limitation of the recovery blocks described here is that they apply only to a single sequential process in isolation. In fact some useful progress has been made towards the recovery of asynchronous co-operating processes, and investigation continues. We hope to publish some preliminary results in this area shortly.

The recovery block concept is only one aspect of a project at the University of Newcastle upon Tyne to investigate computer system reliability. Another, closely related, aspect of this research is the design of highly structured addressing and protection schemes. A proposed ‘recursive virtual machine architecture’ has been documented separately [7]; a simplified version of this architecture is also being incorporated in the emulator.

15. Conclusion

Just as existing protection schemes provide error containment firewalls in space, so the recovery block concept can be thought of as providing firewalls in time. It allows programs containing provisions for error detection and recovery to be structured so as to distinguish clearly between the main program, the acceptance tests, and the action to be taken in error conditions. The provision of successive levels of error recovery permits attempts at purely local error recovery with low overheads, while the errors are contained so that the alternative programs can be run within the same environment as was the main program. The recoverable procedure concept provides a means for the extension of the recovery block concept into more complex situations. These structuring techniques are complemented by a hardware mechanism, whose overheads are in our opinion quite tolerable, which ensures that, for a given choice of recovery blocks, no unnecessary information is saved.

16. Acknowledgements

The development of the recovery block scheme, and the present description of it, has been greatly aided by discussions with colleagues working on or associated with the research project at the University of Newcastle upon Tyne on the Design of Highly Reliable Computer Systems. The funds for this project are provided by the Science Research Council.

References

1. Floyd, R.W., Assigning meanings to programs, in *Mathematical Aspects of Computer Science*, ed. Schwartz, J.T., Amer. Math. Soc. 1967.
2. Randell, B., Operating Systems: the problems of performance and reliability. *IPIP Congress 1971*, Invited Papers, p. 100
3. Balzer, R.M., EXDAMS, Extendible Debugging and Monitoring Systems, *SJCC* 1969, p. 567.
4. Prenner, C.J. et al, An Implementation of Backtracking for Programming Languages, *ACM Annual Conference*, 1972, p. 763.
5. Birtwistle, G.M. et al, *SIMULA Begin*. Student Litteratur/Auerbach 1973.
6. Campbell, R.H. and Habermann, A.N., *The Specification of Process Synchronisation by Path Expressions*. (Published at this conference).
7. Lauer, H.C. and Wyeth D., *A Recursive Virtual Machine Architecture*, TR 54, Computing Laboratory, University of Newcastle upon Tyne, (September 1973) .

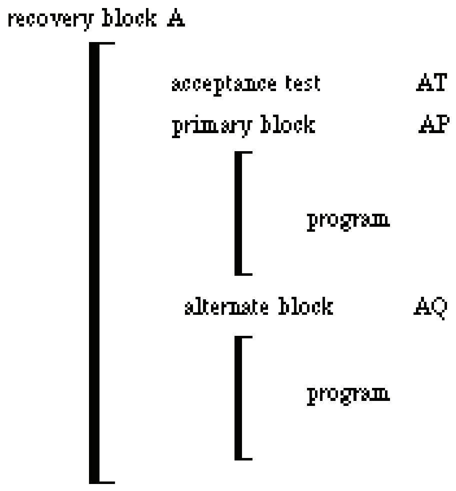

Figure 1.

A diagramatic representation of a recovery block structure.

The primary block corresponds exactly to the block of the Algol-like program and is entered to perform the desired operation.

The acceptance test is executed on exit from the primary block to confirm that the primary block has performed acceptably.

If the primary block is detected to be in error, the alternate block is entered and is required to perform the desired operation in a different way. The acceptance test is then repeated.

```
declare X
recovery block A
    acceptance test     AT
    primary block       AP
        declare Y
            program
        recovery block B
            acceptance test     BT
            primary block       BP
                declare U
                    program
            alternate block     BQ
                declare V
                    program
            alternate block     BR
                declare W
                    program
        program
    alternate block AQ
        declare W
            program
        recovery block C
            acceptance test     CT
            primary block       CP
                program
            alternate block     CQ
                program
        recovery block D
            acceptance test     DT
            primary block       DP
                program
        program
```

Figure 2.
A more complex recovery block structure, showing how the primary and alernate blocks can contain, nested within themselves, further recovery blocks

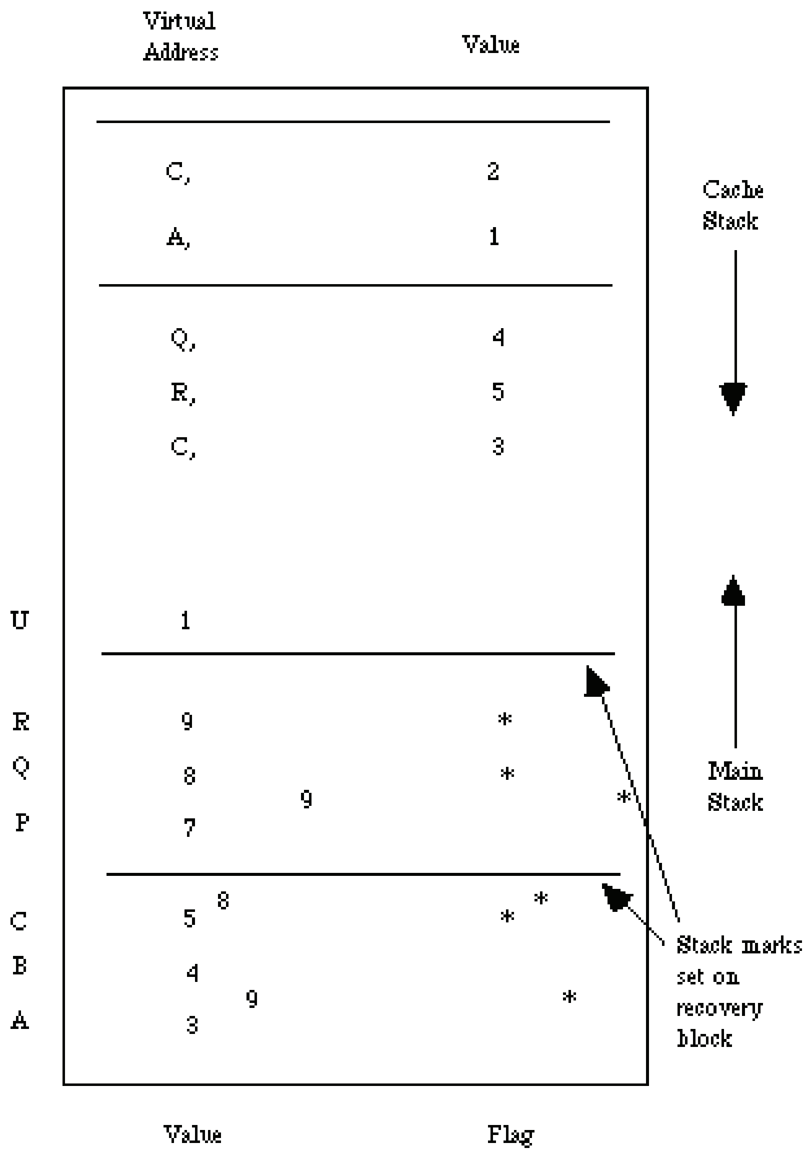

Figure 3.

A diagrammatic representation of teh main stack and cache stack. Assignments to C,Q,R have been made within the current recovery block. Thus these variables have been flagged and their previous values recorded in the cache.

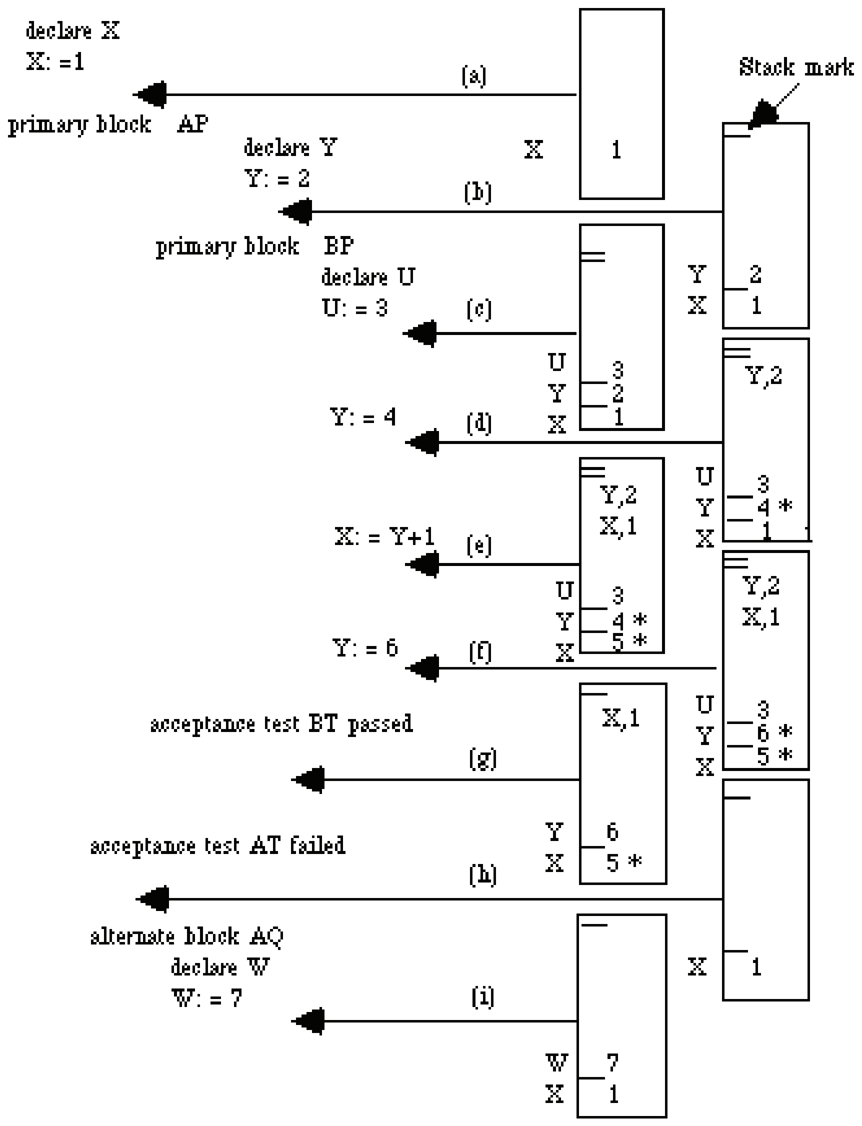

Figure 4.
The states of the stack and the cache during the execution of the program of Figure 2. For descriptions of the various operations depicted see the text, section 9.

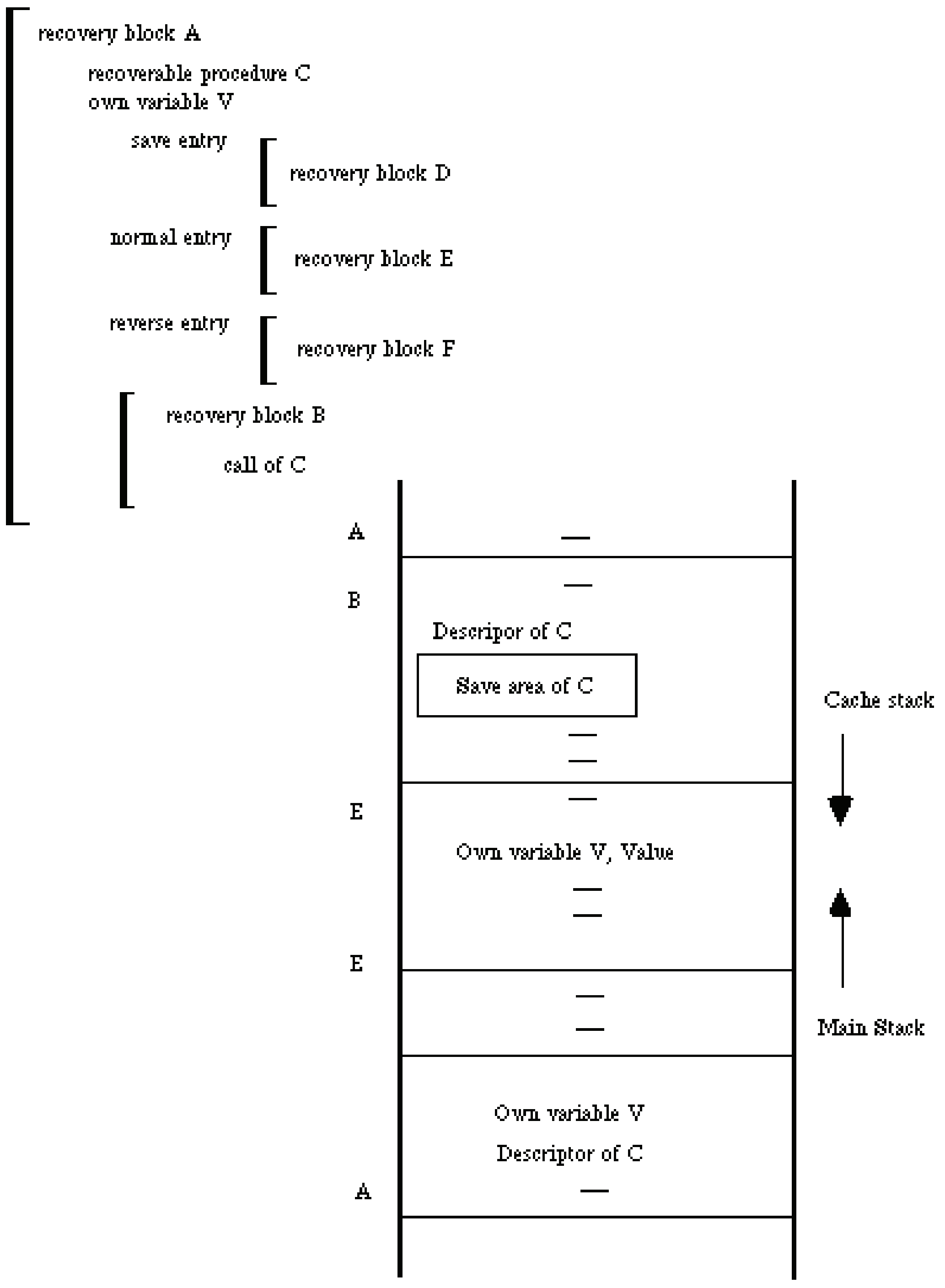

Figure 5.
A program containing a recoverable procedure and a diagramatic representation of the stacks and their contents.